Efficient Processing of Sparse Tensor Decomposition via Unified Abstraction and PE-interactive Architecture

Bangyan Wang, Lei Deng, *Member*, *IEEE*, Zheng Qu, Shuangchen Li,
Zheng Zhang, *Member*, *IEEE*, Yuan Xie, *Fellow*, *IEEE*

Abstract—We propose a novel architecture to efficiently execute sparse tensor decomposition/completion. As the generalization of vectors and matrices, tensors are widely used to process high-dimensional data. It is a natural choice for high-dimensional big data analysis problems in areas such as machine learning and EDA (electronic design automation). Low-rank decomposition is not only an emerging tensor analysis technique but also an effective tool to reduce the storage and computation cost of tensors. Many practical tensors are very sparse, motivating recent studies of **sparse tensor decomposition (SpTD)**. However, conventional general-purpose processors are inefficient to perform SpTD, mainly due to: i) diversity of decomposition algorithms; ii) variable sparsity degree and flexible buffer size requirement; iii) difficulties of fusing multiple execution kernels to pursue better performance.

To solve the above challenges, we propose a unified abstraction for SpTD algorithms and design a specialized accelerator. First, we formulate two types of core kernels (SpLrMM and LrSampling) that serve as a standard form to fit a broad range of SpTD algorithms. Second, we design a sparse tensor engine (STE) to efficiently perform SpTD. STE uses a processing element (PE)-interactive architecture where PEs can be flexibly grouped together via Network-on-Chip (NoC) to share the buffer capacity, bandwidth, and compute resources. This design enables flexible buffer capacity and efficient kernel fusion. By identifying and satisfying three requirements during task mapping, the deadlock issue can be successfully eliminated. We evaluate our accelerator with extensive experiments, and it can achieve an average speedup of 45$\times$ over CPU and 29$\times$ over GPU. The impact of tensor sparsity, PE group size, and memory/compute optimizations are analyzed in detail to give design insights. To the best of our knowledge, this work demonstrates the first accelerator for SpTD, and will stimulate more research on high-performance architecture to facilitate tensor-based data analysis and scientific computing.

1 Introduction

Tensor is the generalization of vectors (i.e., 1D data) and matrices (i.e., 2D data). A d-dimension (termed as mode) tensor of size $n_1 \times n_2 \times \cdots n_d$ can be denoted as $\boldsymbol{\mathcal{X}} \in R^{n_1 \times n_2 \times \cdots n_d}$. If most elements in a tensor are zeros, then the tensor is sparse and can be stored using only $O(nnz)$ rather than $O(n^d)$ memory, where nnz is the number of non-zeros. Today, sparse tensors have been widely used to represent real-world data. For example, Amazon reviews can be represented as a 3-mode User-Product-Review sparse tensor; publications make up a 4-mode Author-Title-Journal/Conference-Year sparse tensor; and any incomplete scientific simulation sweeping over d variables forms a d-mode sparse tensor. Despite the superior expressive power of tensors, the traditional methods based on vectors and matrices are not suited for tensors due to the extra dimensions.

The work was partially supported by National Science Foundation (Grant No. 1817037). Corresponding author: Lei Deng. Bangyan Wang, Lei Deng, Zheng Qu, Shuangchen Li, Zheng Zhang, and Yuan Xie are with the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA (email: {bangyan, leideng, zhengqu, shuangchenli, zhengzhang, yuanxie}@ucsb.edu).

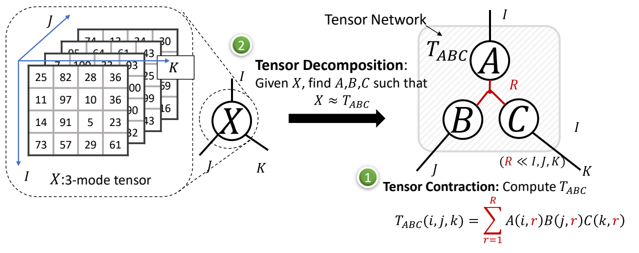


Fig. 1: Tensor contraction and tensor decomposition.

One attractive solution is the **sparse tensor decomposition (SpTD)**. Given a sparse input tensor, usually, an SpTD algorithm can produce a tensor network to approximate the original tensor with much fewer variables, as illustrated in Figure 1. Previous work have already observed that the key data patterns can be extracted by SpTD algorithms, making them very useful in big data analysis, such as social networks analysis [1], [2], [3], discussion tracking [4], [5], Internet traffic analysis for cybersecurity [6], [7], [8], [9], and healthcare [10], [11], [12]. Another important application of SpTD is for the completion of missing data: we can regard the majority of zero entries in a tensor as missing data, and then the tensor network obtained by SpTD can serve as the trained model to predict the missing values [13], [14], [15], [7], [8], [9], [12]. Furthermore, SpTD can significantly compress the data volume and save the computational cost [16], [17], [18]. Despite the wide application scope, the efficient processing of SpTD is not an easy task due to the following difficulties.

Algorithm Diversity. There are many popular methods of tensor decomposition [19], including (but not limited to) CP [20], Tucker [21], tensor train (TT) [22], and hierarchical Tucker (HT) [23] decomposition. Moreover, there are different forms to formulate and solve the basic optimization problem in SpTD, including tensor network structure, loss function, optimization

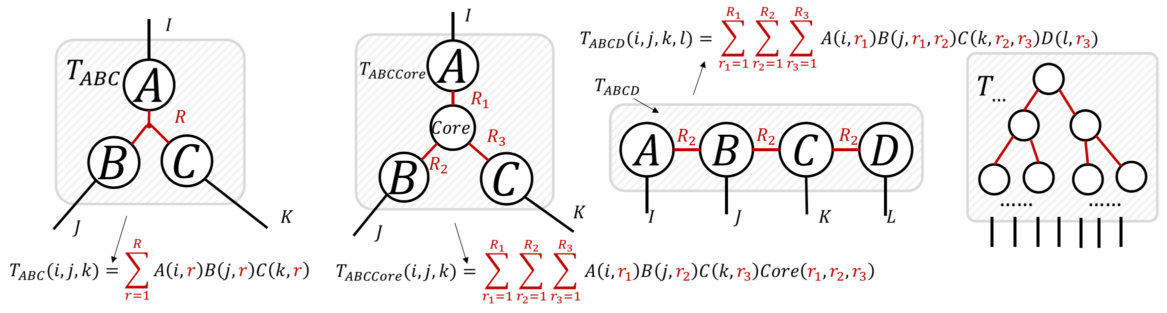

$T_{ABCD}(i,j,k,l) = \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\sum_{r_3=1}^{R_3} A(i,r_1)B(j,r_1,r_2)C(k,r_2,r_3)D(l,r_3)$
$T_{ABC}(i,j,k) = \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r)$
$T_{ABCCore}(i,j,k) = \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\sum_{r_3=1}^{R_3} A(i,r_1)B(j,r_2)C(k,r_3)Core(r_1,r_2,r_3)$
T_{ABC}
$T_{ABCCore}$
T_{ABCD}
$T_{\dots}$
Core

Fig. 2: Different tensor network structures: CP, Tucker, TT (tensor train), and HT (hierarchical Tucker), from left to right.

method, constraint condition, and so forth. Facing such diverse algorithms, quite different execution kernels are required and it is difficult to simultaneously achieve high flexibility and high performance.

Sparsity Variance. In an SpTD algorithm, most execution time is spent on data access with irregular indices. The opportunity of data reuse depends on the buffer capacity and tensor density. Sparser tensors need larger buffer capacity. In real-world cases, the density of a tensor varies in a wide range (e.g. $10^{-9} \sim 10^{-1}$ in the case of 3-mode tensors), while both the local and global buffer size are fixed in conventional architecture. As the sparsity rises, the utilization of the private local buffer gradually decreases. This also increases the requirement for data movement bandwidth if most accesses fall into the shared global buffer or even off-chip memory.

Kernel Fusion Hardness. Fussing multiple execution kernels (following, we simply call it 'kernel') into one kernel is a promising solution to reduce off-chip memory accesses and to eliminate redundant operations. However, executing fused kernel in parallel is hard on traditional architectures, because fussing kernels complicates the write-conflict problem.

These challenges motivate us to develop a top-down solution for the efficient processing of SpTD. On the algorithm side, we propose a unified abstraction via two general sparse tensor kernels to support a broad range of SpTD algorithms. On the architecture side, we design a specialized architecture, sparse tensor engine (STE), to efficiently execute SpTD. STE allows processing elements (PEs) to form tightly collaborated groups to share buffer, bandwidth and compute resources. In this way, we provide a flexible buffer capacity to address the sparsity variance issue, to share the bandwidth of each PE, and to avoid the write conflict. **Our contributions** in this work are summarized as follows:

- We propose a unified abstraction with two general sparse tensor kernels to describe various SpTD algorithms.
- We design an STE to accelerate SpTD. Using a lightweight but effective PE-interactive design, STE satisfies the flexible buffer size requirement and solves the write-conflict problem to enable kernel fusion.
- We conduct extensive experimental evaluations. STE shows an average speedup of 45× over CPU and 29× over GPU. We further analyze the impact of tensor sparsity, PE group size, and memory/compute optimizations in detail.

2 Brief Background

Tensor network is a graph used to represent the multiplication of multiple tensors, which reflects the format of tensor decomposition. Each node in a tensor network denotes a tensor, while the edges connecting to this node represent the tensor's modes (i.e. dimensions). For example, a node representing a vector (i.e. 1D tensor) just has exactly one edge, and a node representing a matrix (i.e. 2D tensor) has two edges, etc. If an edge connects two (or more) tensors (e.g. red lines in Figure 1), it means that the two tensors will be multiplied and summed up along this mode. But if an edge of a tensor does not connect to any other tensor, then this dimension will be preserved. Finally, a tensor is obtained after such multiplication of the entire tensor network. Taking Figure 1 as an example, factor tensors A, B, and C form a tensor network; indices i, j, and k are preserved in T_{ABC} while r is eliminated by summation, following

$$T_{ABC}(i,j,k) = \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r). \tag{1}$$

In practice, a tensor network can have different structures as shown in Figure 2. The above procedure of computing the final tensor (i.e. T_{ABC}) for a given tensor network is called **tensor contraction**. The inverse operation of tensor contraction is called **tensor decomposition**, i.e. finding an appropriate tensor network so that the contraction result can approximate a given tensor. We focus on tensor decomposition in this work. To avoid confusion, in the rest sections, we call the original tensor to be approximated as an **input tensor** (which is sparse in the context of SpTD), while the tensors in the tensor network as **factor tensors**. We regard a tensor/matrix as **low-rank** if it is the contraction result of a tensor network. An easy way to get a sense of SpTD without knowing too much background is to look at an SpTD example in its simplest possible form. For instance, the following is the tensor decomposition problem corresponding to the tensor network illustrated in Figure 1:

$$\arg\min_{A,B,C} \sum_{i,j,k} \left(\mathcal{X}_{i,j,k} - \sum_{r=1}^{R} A(i,r)B(j,r)C(k,r) \right)^2 . \tag{2}$$

3 Challenge and Motivation

As mentioned in Section 1, processing SpTD algorithms on conventional general-purpose architectures is inefficient. In this section, we provide a more detailed analysis of the challenges that motivate our solution.

3.1 Limited Algorithm Flexibility

An important step to accelerate a domain-specific application is to summarize common computation patterns and extract reusable kernels. However, it is challenging in the context of SpTD because of its algorithm diversity. In general, an SpTD algorithm can be written as the following form of optimization problem:

$$\begin{aligned} &\underset{factor\ tensors}{\arg\min} \sum_{i,j,k\in\Omega} E(\boldsymbol{\mathcal{X}}_{i,j,k},\ \boldsymbol{\mathcal{T}}_{i,j,k}), \\ &\qquad \boldsymbol{\mathcal{T}}_{i,j,k} = \mathrm{Contract}(factor\ tensors)_{i,j,k} \\ &\text{s.t.}\quad additional\ constraints \end{aligned} \tag{3}$$

where $\boldsymbol{\mathcal{T}} = \mathrm{Contract}(factor\ tensors)$ computes the predicted tensor by contracting the tensor network, and E computes the approximation error between the predicted $\boldsymbol{\mathcal{T}}$ and the input tensor $\boldsymbol{\mathcal{X}}$. It is clear that SpTD algorithms can have 1) different tensor network structures (reflected by "Contract"), 2) different loss functions to define the approximation error (i.e. E), 3) different constraint conditions, and 4) sometimes different optimization methods to find the solution. Therefore, the kernels required by SpTD algorithms can have huge diversity, as shown in the left side of Figure 3.

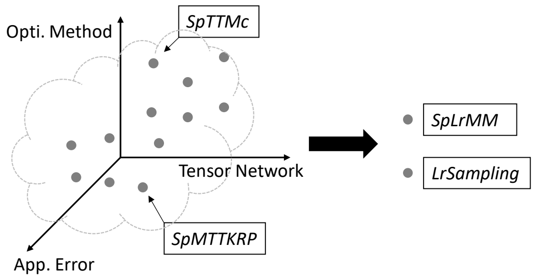


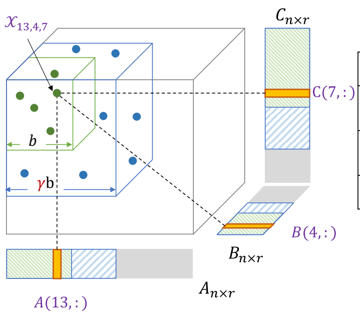
Fig. 3: Algorithm diversity of SpTD (left) and two proposed core kernels to cover most algorithms (right).

Although the existing BLAS library covers part of the required operations of SpTD (such as matrix multiplication and matrix inversion), they cannot handle sparse tensors. To address this issue, some specialized kernels to accelerate the processing of sparse tensors on general processors or distributed platforms are developed, such as SpMTTKRP kernel [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37] and SpTTMc kernel [24], [26], [38], [35], [36], [39], [40], [41]. However, both SpMTTKRP and SpTTMc are restricted to specific SpTD algorithms: SpMTTKRP is mainly used for CP decomposition with square loss; SpTTMc is for Tucker decomposition using HOSVD or HOOI method. In short, they are not general enough.

Requirements: To avoid such a case-by-case study and accelerate the SpTD study in a more general manner, we need to find a unified abstraction for kernels that are required by various algorithms.

3.2 Variable Buffer Size Requirement and Insufficient Data Movement Bandwidth

There is a very unique property about the data access pattern in SpTD algorithms. That is, when processing any sparse entry of $\boldsymbol{\mathcal{X}}$ (e.g. $\boldsymbol{\mathcal{X}}_{ijk}$), the data access is restricted to a very small portion of factor tensors (i.e. $A(i,:)$, $B(j,:)$, $C(k,:)$) determined by the indices (i.e. i, j, k). If two sparse entries $\boldsymbol{\mathcal{X}}_{ijk}$ and $\boldsymbol{\mathcal{X}}_{i'j'k'}$ have part of identical indices, some accesses of factor tensors can be reused. For instance, the accessed data $A(i,:)$ can be reused when requesting $A(i',:)$ if we have $i = i'$. However, the indices of consecutive sparse entries are unpredictable if without careful pre-processing, due to the high randomness. This lack of locality impedes the exploitation of potential data reuse.



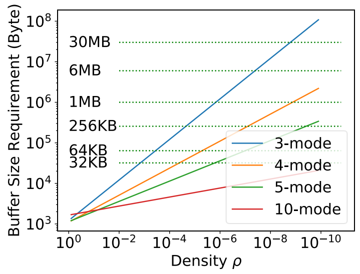
Tilling box size	b	γb
Non-zero sparse entries (nnz)	ρb^d	$\rho\gamma^d b^d$
Required Buffer Size	$O(b)$	$O(\gamma b)$
Data Reuse Rate	$O(\rho b^{d-1})$	$O(\rho\gamma^{d-1} b^{d-1})$

Fig. 4: The data reuse analysis in SpTD.

To maximize the data reuse between sparse entries, the sparse entries close to each other should be processed together, as shown in Figure 4. For example, we can process the sparse entries in a tiling box with each dimension being b. Assuming the density of sparse tensor $\boldsymbol{\mathcal{X}}$ is ρ, then, totally about ρb^d sparse entries within this tilling box need to load only $O(b)$ instead of $O(\rho b^d)$ amount of data due to inter-sparse entry data reuse. The only constraint is that buffer size C_{buffer} should be big enough to hold the $O(b)$-size factor tensor data, i.e. $b \propto C_{buffer}$. With this knowledge, data reuse ratio λ can be speculated as

$$\lambda \propto \frac{\rho O(b^d)}{C_{buffer}} \propto \rho O(b^{d-1}) \propto \rho C_{buffer}^{d-1}. \tag{4}$$

This property leads to a challenge: when the tensor density ρ decreases (i.e. sparsity increases), the data reuse rate drops proportionally. Unfortunately, ρ in real-world tensors usually has a wide range, such as from 10^{-1} to 10^{-9}, causing an unbearably low data reuse rate. However, this property also brings an **opportunity**: if the buffer capacity increases by a factor of γ, the data reuse rate increases by a factor of γ^{d-1}, which can mitigate the negative effect of the decrease of ρ.



		CPU	GPU
Local Buffer	N	28 cores	80 SMs
	C	1056 KB	384 KB
	B	140~360 GB/s	172~2,457 GB/s
	L	4~13 cycles	28 cycles
Global Buffer	C	38.5 MB	6 MB
	B	40.5 GB/s	27 GB/s
	L	50~70 cycles	193 cycles
Off-Chip Mem.	B	5 GB/s	11.25 GB/s

Fig. 5: Variable buffer size requirement affected by the tensor density ρ. Here we take the CP decomposition with $R = 8$ as an example. The table lists the number of cores/SMs (N), local memory size per core/SM or memory size of global buffer (C), buffer bandwidth (B), and local buffer hit latency (L). The global buffer bandwidth values are evenly distributed onto each core/SM.

The unique property of SpTD in Equation (4) causes a variable buffer size requirement if we want to achieve a satisfactory data reuse rate as the input tensor sparsity varies, which is illustrated in Figure 5. Whereas, traditional general-purpose architectures that feature "fixed small local buffer & large global buffer" look embarrassing in this situation. Specifically, the required buffer size often exceeds the size of local buffers, while the global buffer, although larger, provides much lower bandwidth than the local buffers. Besides, the local buffers cannot be borrowed to become a part of the global buffer to provide a bigger capacity when necessary.

Requirements: Above problems post two requirements in the architecture design: i) it should allow a flexible buffer configuration to fit different tensor sparsity; ii) it should overcome the limitation of on-chip interconnection bandwidth.

3.3 Difficult Kernel Fusion

When we have multiple kernels to run, we often have a chance to achieve higher efficiency by fusing them into one. There are two benefits: i) part of data for multiple kernels can be loaded only once; ii) part of the computation result can be shared by different kernels. Nevertheless, the kernel fusion is difficult for parallel execution due to the conflict of memory writes from different kernels: we first notice that after the kernel fusion (as in Algorithm 3), each sparse entry in the input tensor (e.g. $\mathcal{X}_{ijk}$) needs to add some results onto multiple objects such as $Out_1(i,:)$, $Out_2(j,:)$, and $Out_3(k,:)$. When processing multiple sparse entries in parallel, such operations may lead to write conflict when any one of the three indices i, j or k overlap between any two sparse entries.

There are several software-level attempts to avoid write conflict in the context of kernel fusion, but none of them can work well in our context. In particular, conflict-free partition [33], [32] divides the sparse entries into different threads (or PEs) such that their indices (e.g. i, j, k, etc.) are simultaneously disjoint. Whereas, it will incur not only over-partition of the workload (i.e. P^d partitions for P threads) but also complicated scheduling and synchronization. Even worse, any conflict-free partition method will inevitably prevent buffer sharing between threads, which is vital for SpTD because a large buffer capacity is needed in the case of high sparsity.

Requirements: Due to the difficulty in solving the write conflict problem at the software level, we need to address it from the architecture level, i.e. it should be able to support "atomic-add"-like operations in large volume.

4 ALGORITHM ABSTRACTION

To address the algorithm diversity problem, we first propose two general core kernels which are parameterizable with respect to the tensor network structure, termed as SpLrMM and LrSampling, and then we show that they are enough to cover a variety of SpTD algorithms.

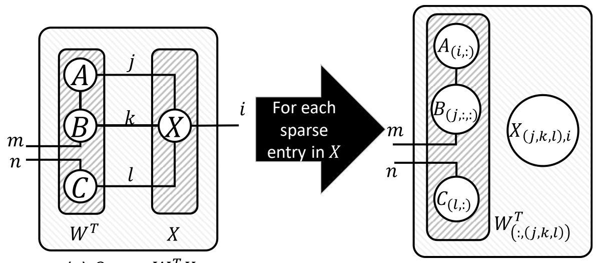
4.1 SpLrMM Kernel

SpLrMM (sparse-matrix low-rank-matrix multiplication). This kernel is a special case of matrix-matrix multiplication, while one of the two operand matrices is sparse and the other is low-rank. It can be simply denoted as

$$Out = W^T X \tag{5}$$

where X is sparse and W is low-rank. The low-rank matrix W can be expressed by a tensor network. An example is given in Figure 6(a), where W has three internal factor tensors: A, B, and C. By varying the network structure inside W, different forms of SpLrMM kernels can be constructed to support different SpTD algorithms. The prior SpMTTKRP and SpTTMc are just two special instances of SpLrMM.

Two unique properties of W and X will be exploited to perform SpLrMM efficiently, i.e. the low-rankness of W and the sparsity of X. The former property allows us to compute the result from the tensor network representation without explicitly constructing W; the latter property allows us to only process the non-zero entries. Let's continue to use the example in Figure 6(a) to show how to exploit these properties. Each entry in the sparse matrix X can be processed separately, causing a partial sum to be added into the corresponding column of Out. It is illustrated in Figure 6(b), equivalent to



(a) $Out = W^T X$ (b) $W^T_{(:,(j,k,l))} X_{((j,k,l),i)}$

Fig. 6: An example of performing SpLrMM.

$$Out_{(:,i)} \leftarrow Out_{(:,i)} + W^T_{(:,(j,k,l))} X_{((j,k,l),i)}. \tag{6}$$

Algorithm 1: Pseudocode for SpLrMM

```
for $No. \in \{1,2,...,nnz\}$ do
    $x \leftarrow$ Value($No.$), $i,j,k,l \leftarrow$ Index($No.$)// $x = \mathcal{X}_{i,j,k,l}$
    $a \leftarrow A(j,:)$, $b \leftarrow B(k,:,:)$, $c \leftarrow C(l,:)$
    $t_1 = ax$ (vector-scalar), $t_2 = t_1 b$ (vector-matrix)
    $t_3 = c^T t_2$ (vector outer product)
    $Out(:,i) \leftarrow Out(:,i) + reshape(t_3)$
end
```

For sparse entry $\mathcal{X}_{i,j,k,l}$, the column $W^T_{(:,(j,k,l))}$ only relies on $A(j,:)$, $B(k,:,:)$, and $C(l,:)$, and the result is accumulated onto $Out(:,i)$. In essence, all data accesses depend exactly on the indices of sparse entries. The pseudo code for SpLrMM is provided in Algorithm 1, corresponding to Figure 6.

Notice that we cannot perform SpLrMM using common matrix-matrix multiplication methods despite being mathematically correct, because W and X have dimension exponentially proportional to the number of factor tensors. Explicitly constructing them for direct matrix-matrix multiplication can easily consume a storage space larger than TBs or even PBs and cause a huge amount of wasted computation. Making use of the sparsity and low-rankness as we did above avoids such a problem.

4.2 LrSampling Kernel

LrSampling (sampling elements from a low-rank tensor according to sparse entries). This kernel is to sample values (e.g. $\mathcal{T}_{i,j,l}$) from a low-rank tensor $\mathcal{T}$ (implicitly represented by a tensor network) given a set of sparse entries ($\forall(i,j,k) \in \Omega$). Similar to SpLrMM, LrSampling is also parameterizable with respect to the tensor network structure. LrSampling can be formulated as

$$Out = \mathcal{T} \circ \Omega \tag{7}$$

where $\circ$ denotes element-wise product and Ω has binary values to indicate the sampling locations. The pseudocode for LrSampling is given in Algorithm 2. Again we see the same access pattern depending on the indices of sparse entries.

TABLE 1: Mapping SpTD algorithms onto the proposed SpLrMM and LrSampling kernels. The colored data are low-rank tensors and sparse tensors, respectively.

Loss Function	Iterative Method	Update Formula Derivation	Kernel Mapping
Square Loss: $\|X-WH\|^2$	ALS	$H \leftarrow (\lambda I + W^T W)^{-1} W^T X$ Key Components: $W^T W$, $W^T X$	*SpLrMM*: $W^T X$ Tensor Contraction: $W^T W$
	Gradient Descent	$\delta H \leftarrow W^T X - (W^T W) H$ Key Components: $W^T W$, $W^T X$	*SpLrMM*: $W^T X$ Tensor Contraction: $W^T W$
	Multiplicative Update	$H_{ai} \leftarrow H_{ai}[(W^T X)_{ai}/(W^T W H)_{ai}]$ Key Components: $W^T W$, $W^T X$	*SpLrMM*: $W^T X$ Tensor Contraction: $W^T W$
Masked Square Loss: $\|\Omega_X \circ (X-WH)\|^2$	ALS	$H[:,i] \leftarrow (\lambda I + W_i'^T W_i')^{-1} W_i'^T X[:,i]$ where $W_i' = \text{diag}(\Omega_X[:,i])W$ Key Components (raw): $W_i'^T W_i'$, $W_i'^T X[:,i]$ Key Components (simplified & batched): $(W^T \odot W^T)\Omega_X$, $W^T X$	*SpLrMM*: $(W^T \odot W^T)\Omega_X$ *SpLrMM*: $W^T X$
	Gradient Descent	$\delta H[:,i] \leftarrow W_i'^T (X[:,i] - W_i' H[:,i])$ where $W_i' = \text{diag}(\Omega_X[:,i])W$ Key Components: $R_1 \leftarrow X - \Omega_X \circ (WH)$, $R_2 \leftarrow W^T R_1$	*LrSampling*: $R_1 \leftarrow X - \Omega_X \circ (WH)$ *SpLrMM*: $R_2 \leftarrow W^T R_1$
	Multiplicative Update	$H_{ai} \leftarrow H_{ai}[(W_i'^T X[:,i])_a/(W_i'^T W_i' H[:,i])_a]$ Key Components (raw): $W_i'^T W_i'$,$W_i'^T X[:,i]$ Key Components (simplified & batched): $(W^T \odot W^T)\Omega_X$, $W^T X$	*SpLrMM*: $(W^T \odot W^T)\Omega_X$ *SpLrMM*: $W^T X$
KL Divergence: $D_{KL}(X\|WH)$	Multiplicative Update	$H_{a\mu} \leftarrow H_{a\mu}[(\sum_i W_{ia} \frac{X_{i\mu}}{(WH)_{i\mu}})/(\sum_i W_{ia})]$ Decompose into two steps: Step 1: $S_{i\mu} \leftarrow [X_{i\mu}/(WH)_{i\mu}]$ Step 2: $H_{a\mu} \leftarrow H_{a\mu}[(W^T S)_{a\mu}/(W^T \vec{1})_a]$ Key Components: $[X_{i\mu}/(WH)_{i\mu}]$, $W^T S$, $W^T \vec{1}$	*LrSampling*: $[X_{i\mu}/(WH)_{i\mu}]$ *SpLrMM*: $W^T S$ Tensor Contraction: $W^T \vec{1}$

Algorithm 2: Pseudocode for LrSampling

```
for $No. \in \{1,2,...,nnz\}$ do
    $i, j, k \leftarrow$ Index$(No.)$
    $a \leftarrow A(i,:)$, $b \leftarrow B(j,:,:)$, $c \leftarrow C(k,:)$
    $t_1 = ab$ (vector-matrix), $t_2 = t_1 c^T$ (vector inner product)
    $Out(No.) \leftarrow t_2$ // $Out(i, j, k) \leftarrow t_2$
end
```

Algorithm 3: A pseudocode example for the kernel fusion

```
for $No. \in \{1,2,...,nnz\}$ do
    //Refer to Algorithm 1 and Algorithm 2 for the data, operation,
      result required by SpLrMM and LrSampling, respectively
    Load:
    $x \leftarrow$ Value$(No.)$, $i, j, k, l \leftarrow$ Index$(No.)$// $x = \mathcal{X}_{i,j,k,l}$
    Load factor tensor data for LrSampling
    Load factor tensor data for SpLrMM
    Load factor tensor data for $\cdots$
    Compute:
    Perform operations of LrSampling
    Perform operations of SpLrMM
    Perform operations of $\cdots$
    Update:
    Update the result of LrSampling
    Update the result of SpLrMM
    Update the result of $\cdots$
end
```

4.3 Connection to SpDT Algorithms

To show that the proposed SpLrMM and LrSampling kernels are able to cover a variety of SpTD algorithms, we first rewrite the objective function of an SpTD algorithm into an equivalent matrix form. For example, the original objective function such as $argmin_{A,B,C} \|\mathcal{X} - \mathcal{T}_{A,B,C}\|$ can be modified as $argmin_{A,B,C} \|X - W_{B,C} H_A\|$, where H_A and X are matrices reshaped from A and $\mathcal{X}$, and the matrix $W_{B,C}$ is obtained by removing the A's component from the original tensor network $\mathcal{T}_{A,B,C}$. After this transformation, we can obtain an update formula for H_A (and equivalently, for A) from the matrix form. Such updates can often be implemented using SpLrMM and LrSampling kernels, as exemplified in Table 1.

The same procedure is also applicable to update other factor tensors such as B and C. In this way, the tensor factors are updated repeatedly until convergence. In each update, there are usually multiple SpLrMM/LrSmpling kernels and each of them is a loop iterating over all sparse entries (See Algorithm 1, 2). We can fuse those separate kernels (or loops) into one to reduce redundant computation and data access, as shown in Algorithm 3. With the above descriptions, it can be seen that our proposed kernels are general enough to support a wide range of SpTD algorithms and are in the standard forms accepted by our architecture.

5 STE Architecture Design

We introduce an efficient architecture, STE, to run the mentioned core kernels in this section. Figure 7 depicts the overview of our design. There are 16×16 PEs in total, a scheduler, and a memory controller. Multiple PEs can interact with each other through a Network-on-Chip (NoC) infrastructure to form a PE group for task processing. We will introduce the design of major components, describe the task execution flow, and explain how to deploy SpTD algorithms.

5.1 Design Philosophy

First, let's focus on the challenge of the requirement for flexible buffer size. Each PE has a fixed 128KB private buffer. When the demanded buffer size exceeds 128KB, we group a flexible number of PEs into a tightly collaborated group to share their buffer capacity so that their total capacity is $PE_{Num} \times 128$KB where PE_{Num} is the PE group size. Whereas, simply grouping PEs cannot solve the problem completely. Accessing data from other PEs through a longer-latency and lower-bandwidth NoC is still far slower than accessing a PE's local buffer. To address this issue, we propose to further share the compute resources of PEs within a group. Specifically, we actively transfer tasks from one PE to

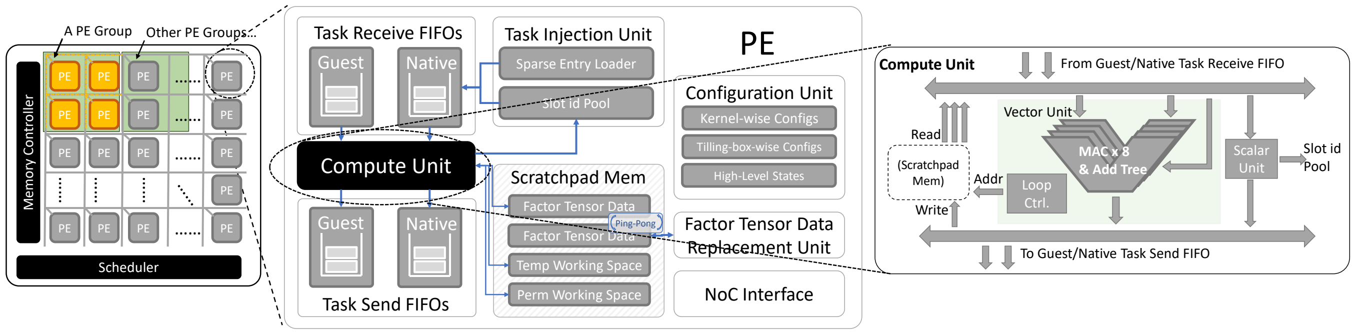


Fig. 7: Overview of the STE architecture design.

another that has the expected data. The solution of task traveling saves a lot of traffic on the NoC. For example, when performing a matrix-vector multiplication, if the matrix is in one PE while the vector is in another, transferring the task to the PE with the matrix needs only a vector movement. Eventually, the computation can enjoy both the high bandwidth of intra-PE private buffers and the large capacity of the inter-PE group.

Second, let's look at the challenge of write conflicts in kernel fusion, which requires the support of atomic-add-like operations with high throughput. With the PE-interactive design, it is now solved for free. The local buffer is still private to each PE and other PEs can only modify it by transferring the task to the current PE. In this way, write conflicts will not happen.

At last, several additional problems must be solved to make above idea practical. For example, we need to minimize the communication/control overhead of such task travel design, properly allocate the working space for tasks given a limited on-chip memory space, avoid idleness and deadlock when PEs transfer tasks to each other. All these problems are solved by our STE, the architectural details of which are provided as follows.

5.2 Processing Element Specialization

A PE contains a configuration unit, task receive/send FIFOs, a task injection unit, a factor data replacement unit, a scratchpad memory, a compute unit, and an NoC interface, as illustrated in the right side of Figure 7.

Terminology definitions. A task will traverse multiple PEs through the NoC. The PE where it was created is the home PE and all other PEs are remote PEs. When the task is in its home PE, it is viewed as a native task, otherwise, it is treated as a guest task. Note that a remote PE for one task might be the home PE for another task in the meantime.

Configuration Unit. The configuration unit maintains the PE's high-level states and exposes the configuration registers and the instruction buffer to the scheduler.

Task FIFOs. There are two pairs of task receive/send FIFOs: one for native tasks and the other for guest tasks. The arriving task will be put into one of the two task receive FIFOs depending on its status (being native or guest). After processed by the compute unit, the task will be put into the corresponding task send FIFO (native or guest). The separation of native FIFOs and guest FIFOs is intended for deadlock avoidance when congestion happens, which will be discussed later. Task FIFOs enable PE to communicate with other PEs using messages.

Task Injection Unit. This unit loads sparse entries in COO (coordinate list) format from off-chip memory, creates one task for each sparse entry, and injects the task into the native task receive FIFO. The task injection unit should not keep on injecting new tasks without considering whether old tasks have been finished, in other words, it should be self-throttling. This is realized by designing a *slot id* pool and bounding the maximum number of on-the-fly tasks launched by each PE. Each time a task is injected, a *slot id* will be granted to the task, which will be returned to the injection unit when the task is destroyed (this implies the last stage of a task must be processed in its home PE). The initially available number of *slot id*s in the pool is $SlotMax$ (i.e. the mentioned bound) and no new task is allowed to be injected when the pool is empty.

Scratchpad Memory. The scratchpad memory in each PE is responsible for storing a part of factor tensor data and also storing the intermediate results as a working space. The working space has two types: temporary (Temp) one and permanent (Perm) one. The Temp working space can be used by both native and guest tasks. When the task leaves this PE, the Temp working space is available to the following tasks and can be overwritten. In contrast, the Perm working space is only visible to native tasks of the current PE, which requires the *slot id* for data access. This is the reason that it is divided into $SlotMax$ separate chunks. Every on-the-fly task has one such safe box in the Perm working space of its home PE to store data even if the task has left its home PE. The data will safely remain until the task is finished and the *slot id* is returned to the task injection unit. With the Perm working space, the carried intermediate data in the NoC can be reduced since only the required data will be transferred while the rest remain in the home PE.

Factor Tensor Data Replacement Unit. Factor tensor data replacement unit prefetches factor tensor data of next tiling box into the ping-pong buffer. It actually helps decouple the off-chip access and on-chip computation. Note that each PE in a group only loads its own part of factor tensor data.

Compute Unit. This unit supports five types of instructions: i) taking a message from task receive FIFOs and jumping to the correct PC (program counter); ii) figuring out which PE has the expected factor tensor data (e.g. for $A(i,:)$, which is $mod(i, PE_{NUM})$); iii) performing arithmetic operations; iv) creating a message and putting it into task send FIFOs; v) returning the *slot id* to the task injection unit when the task is finished.

The arithmetic operations include scalar ones (e.g. $+$, $-$, $\times$, $\div$) and tensor ones (e.g. inner product ($\cdot$), outer product ($\otimes$), element-wise product ($\circ$), matrix-vector product), which are able to support the operations required by SpLrMM (Algorithm 1) and LrSampling (Algorithm 2). We therefore use two techniques to improve the throughput. First, these operations need only one instruction to do all jobs rather than using software loops. The instruction will be expanded by hardware to generate a sequence of micro operations, which avoids potential instruction overhead. Second, these operations use an 8-lane vector unit to exploit the data-level parallelism.

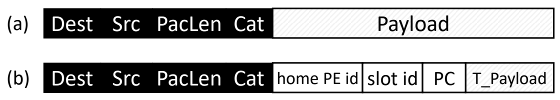


Fig. 8: Packet format of task messages: (a) general format; (b) specific format for inter-PE communication. The black segments will be processed by the NoC interface.

NoC Interface. The NoC interface receives/sends packets from/to other PEs. The packet format is shown in Figure 8(a). The fields are destination (*Dest*), source (*Src*), packet length (*PacLen*), traffic category (*Cat*), and *Payload*. The packets between PE$\leftrightarrow$PE, PE$\leftrightarrow$scheduler, PE$\leftrightarrow$memory controller are classified into different traffic categories.

The packet format for the PE$\leftrightarrow$PE communication is given in Figure 8(b). The additional head for *Payload* is very lightweight: *home PE id* (8 bits), *slot id* (8 bits), and *PC* (16 bits). *PC* indicates the progress of this task. *T_Payload* carries the required data of the destination PE, in which the length and the content vary across processing stages.

5.3 The Lifetime of A Task

From the two general core kernels in Algorithm 1 and 2, we can see that for each sparse entry, a few operations need to execute as listed in the loop body. We create one task for every sparse entry, so totally *nnz* tasks will be created for each kernel execution. Then, we explain the lifetime of a task through a simplified example following Algorithm 1. Assume that the sparse entry is $\boldsymbol{\mathcal{X}}_{4,7,2,5} = 3.5$ and the operations needed for this sparse entry are 1) multiplying $\boldsymbol{\mathcal{X}}_{4,7,2,5}$ with vector $A(4,:)$ to get vector t_1; 2) multiplying vector t_1 with matrix $B(7,:,:)$ to get vector t_2; 3) compute outer product of the vector t_2 and vector $C(2,:)$ to get matrix t_3; 4) reshaping and accumulating matrix t_3 onto $Out(:,5)$. Notice that $A(4,:)$, $B(7,:,:)$, $C(2,:)$, and $Out(:,5)$ may be in different PEs within a group. For instance, we assume $A(4,:)$ is in the PE whose $id = mod(4, PE_{Num})$; $B(7,:,:)$, $C(2,:)$, and $Out(:,5)$ are similar. Therefore, the task needs to travel several PEs to finish the execution.

In the beginning, one PE (can be any PE in the group, e.g. PE0) loads the integer indices 4,7,2,5 and its value 3.5, and then creates a task. PE0 now becomes the home PE of the created task. Task Injection Unit grants a *slot id* (e.g. 3) to this task and puts it into Native Task Receive FIFO as a message. The format is shown in Figure 8, and the initial value of *PC* is 0. Compute Unit iteratively pulls tasks from Task Receive FIFOs to execute. Once a task is pulled and executed, it starts at the instruction specified by *PC* in the message (the list of instructions is the same in all PEs). Instructions are executed until a “send task” instruction is met, and then a message in Task Send FIFOs is created to send this task away and tell Compute Unit to start working on the next task in Task Receive FIFOs. In the above example, the task will start at PE0, and then it will go through four steps: 1) moving to the PE that has $A(4,:)$, performing operations there and moving back to PE0; 2) moving to the PE that has $B(7,:,:)$), performing operations there and moving back to PE0; similarly, 3) and 4) for $C(2,:)$ and $Out(:,5)$, respectively. Finally, when it moves back to PE0, the task returns the *slot id* to Task Injection Unit and ends.

5.4 Deadlock Avoidance

Congestion may happen if some PEs become hot spots. If their receive FIFOs overflow, the traffic jam will diffuse into the NoC and further blocks other traffics. We notice that, although a PE can have maximally $(PE_{Num} - 1) \times SlotMax$ guest tasks coming at the same time in the worst case, it has no more than $1 \times SlotMax$ native tasks at anytime. This inspires us to satisfy additional requirements to avoid deadlock: i) Tasks must alternately visit home PE and remote PE, i.e. home PE$\rightarrow$remote PE$\rightarrow$home PE$\rightarrow$remote PE$\rightarrow$...; ii) The native task receive FIFOs should never overflow by setting the FIFO capacity larger than $Size(packet) \times SlotMax$; iii) The home PE to the remote PE (going out) traffic should never block the path from the remote PE to the home PE (coming back) traffic, even if the former itself is blocked, which can be achieved by allocating a set of virtual channels in the NoC to the latter path.

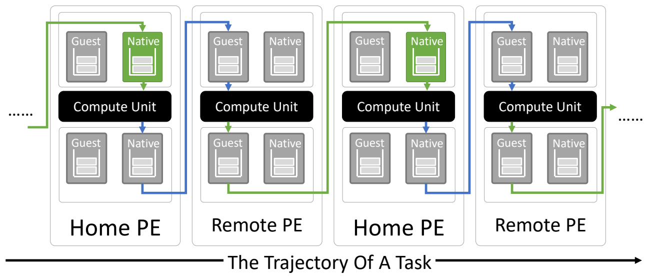


Fig. 9: The task trajectory across PEs.

If all the above requirements are satisfied, the deadlock will not happen. As Figure 9 shows, since the native task receive FIFO will never overflow neither block on the road, the traffic from the remote PE to the home PE (green arrows in Figure 9) will not be permanently blocked. This means that the guest send FIFO can always send packets and thus reflecting that the compute unit can continuously process guest tasks, which similarly indicates that the guest receive FIFO will never be blocked. Therefore, the traffic from the home PE to the remote PE (blue arrows in Figure 9) is also smooth. In a nutshell, the deadlock can be avoided.

5.5 Mapping Algorithms onto STE

In this section, we discuss how to map SpTD algorithms onto STE, which includes algorithm compilation, data preprocessing, and hardware execution, as given in Figure 10.

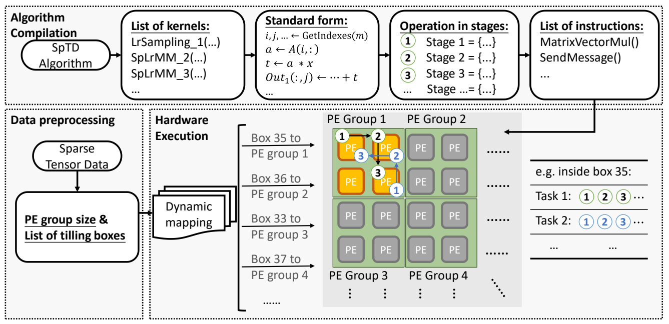


Fig. 10: Algorithm mapping flow.

Algorithm Compilation. We start with the SpTD algorithm by expressing it using SpLrMM and LrSampling kernels. Then, we fuse these kernels into a standard form as in Algorithm 3. Next, we group the operations in the standard form into multiple processing stages, so that a task will execute one stage on one PE.

The intermediate results are relayed between stages via the Perm working space or via the T_Payload field of the task. The later should be minimized to reduce on-chip communication overheads. To do this, we first assign the factor tensor data access operations like $a \leftarrow A(i,:)$ and $b \leftarrow B(j,:,:)$ into different stages. Then, we assign the compute operations like $t \leftarrow a*b$ to the most correlated stage that will incur the least data movement. For example, if $Size(a) < Size(b)$, we will assign $t \leftarrow a*b$ to the stage with $b \leftarrow B(j,:,:)$ and transfer a over the NoC. After all operations for a sparse entry are assigned into appropriate stages, we insert the "send message"-related instructions between every two stages, such as determining the next PE id and packaging data into the message payload. Finally, we represent the program as a list of instructions.

Data Preprocessing. The sparse input tensor will be tiled into multiple boxes of $b_1 \times b_2 \times \cdots \times b_d$ size, and each tiling box will be processed in a PE group. The selection of $\{b_i\}$ follows two principles (refer to Section 3): i) Enlarging the tiling box to improve the data reuse rate if the tensor density ρ is low; ii) Without violating the first principle, always keeping the tiling box as small as possible to reduce the required buffer size (leading to a smaller PE group with less on-chip traffic). With a tiling box size given, the PE group size is then determined and the sparse entries in each tiling box are packed as a list of tasks.

Hardware Execution. The scheduler will dynamically map tilling boxes onto PE groups. The PE groups run independently, while the PEs within each group interacts with each other. All PEs need to load the required factor tensor data into the scratchpad memory before the computation starts. Fortunately, the load latency can be hidden using a ping-pong buffer. Then, the PEs can start working on the sparse entries. Once all sparse entries in a tiling box are processed, the scheduler switches the PE group to the next tiling box.

5.6 Minor Optimizations

Relative Indexing. After tensor tiling, the sparse entries just need to store the relative indices within each tilling box. This results in reduced index overhead and helps to reduce the off-chip memory accesses.

Continuous Tiling Box Assignment. The scheduler always attempts to assign adjacent tiling boxes to the same PE group. In this way, only one piece of factor tensor data (e.g. $A(i,:)$) needs to be replaced, and the rest $d-1$ pieces can be continuously used (e.g. $B(j,:)$, $C(k,:)$). This also reduces the off-chip memory accesses.

Random Permutation. To achieve inter-PE workload balance, we adopt three extra steps during prepossessing: i) Before tiling, we randomly permute the indices of each tensor mode; ii) After tiling, we assign the sparse entries of each tiling box evenly to intra-group PEs; iii) Finally, the order of assigned sparse entries in each PE is randomly permuted.

5.7 Advantage Summary

Major Advantages. i) *Algorithm Generality:* STE can support a wide range of SpTD algorithms via the unified abstraction. ii) *Flexible Buffer Capacity and High Bandwidth:* Our architecture meets the requirement of variable buffer size by flexibly adjusting the PE group size based on the actual tensor density ρ. Meanwhile, thanks to the task transfer, tasks can enjoy the high bandwidth of local scratchpad memory inside each PE it traverses. iii) *Kernel Fusion Enabled:* Our architecture is write-conflict-free because the compute transfer allows the operation taking place locally in the PE that stores the required data.

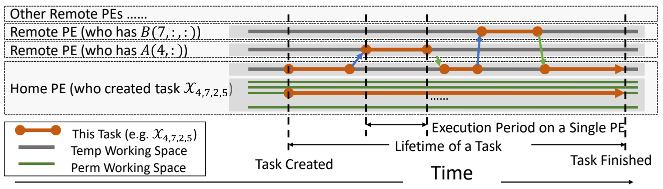

Other Remote PEs ......
Remote PE (who has $B(7,:,:)$)
Remote PE (who has $A(4,:)$)
Home PE (who created task $\mathcal{X}_{4,7,2,5}$)
......
This Task (e.g. $\mathcal{X}_{4,7,2,5}$)
Temp Working Space
Perm Working Space
Execution Period on a Single PE
Lifetime of a Task
Task Created
Task Finished
Time

Fig. 11: The memory trajectory across PEs.

Other Advantages. i) *Latency Insensitivity:* First, all the memory accesses during running are local and there is no performance loss caused by the stall on non-local memory accesses. This is because if a task needs to access factor tensor data that are not on the current PE, the task will migrate itself to another PE which has the expected data via the NoC (see Figure 11), leaving the compute unit to the next task to maintain full utilization. The overall throughput will not be degraded if we can always ensure that every receive FIFO is not empty and the compute unit is busy. ii) *Lightweight Message:* Each message only includes 4B extra overhead (i.e. *home PE id*, *slot id*, and *PC*) beyond the effectual payload. Moreover, the Perm working space in the home PE reduces the length of *T_Payload* because only the data required by destination PE will be transferred.

6 Evaluation

6.1 Experimental Setup

System Configuration and Simulation. The performance is evaluated using two levels of simulation: single PE level and PE array level. In the single PE level, we implement send/receive FIFOs and the compute unit using Chisel [42]. With the RTL simulation, we validate the functional correctness and completeness of our instructions, and measure their cycle-accurate execution time. To obtain the overall performance, we modify an NoC simulator (i.e. BookSim2.0) and simulate the PE array. We add modules to reproduce the PE's behaviors including the task injection unit, task receive/send FIFOs, the compute unit, and the factor data replacement unit. The compute unit's behaviors are simplified to a countdown timer in which the delay is calculated according to the RTL simulation result of a single PE. The simulated traffic in BookSim2.0 includes both PE-to-PE and PE-to-memory communications. To capture more insights, we add extra performance counters to measure the utilization of the compute unit, the NoC, and the memory controller.

TABLE 2: System configuration.

PE	PE array size: 16×16 Scratchpad Memory: 128 KB per PE Task Send/Receive FIFOs: 16 KB per PE Compute Unit Vector Width: 8 *SlotMax*: 50
NoC	Channel Width: 128 bits
Off-chip Memory	Bandwidth: 24∼96 GB/s

TABLE 3: Area and power breakdown at 22nm and 1GHz.

	Interconnect	Compute	Scratchpad	Memory Controller	Total
Power	26%	29%	42%	3%	62 W
Area	43%	28%	28%	1%	115 mm^2

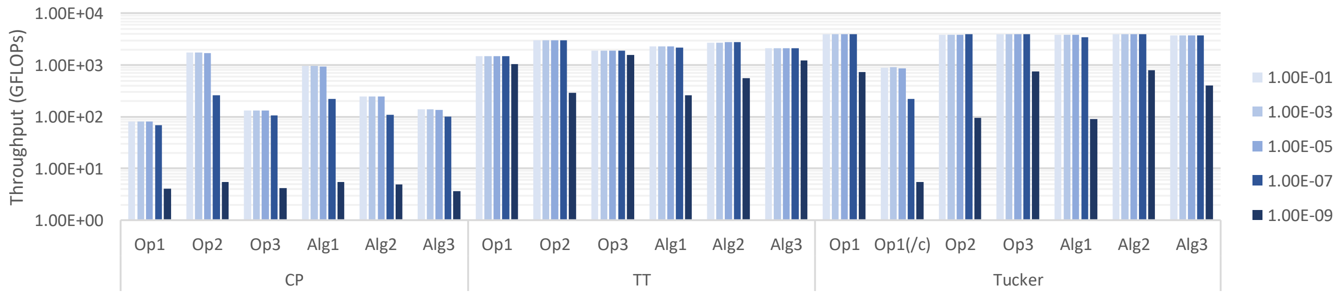


Fig. 12: STE performance for different algorithms under various input tensor density. Here the PE group size is set to 8×8.

To estimate the area and power, we further use CACTI [43] to simulate the SRAM, DSENT [44] to simulate the NoC, and McPAT [45] to simulate the DRAM memory controller. The rest parts of PE are implemented in RTL and compiled using Synopsis Design Compiler. For the floating-point unit, we leverage the implementation of the Berkeley FPU [46]. We use 22nm technology library in above simulations to make a more fair comparison with modern CPU/GPU. Table 2 presents the system configuration and Table 3 shows the power and area breakdown. Please note that the default off-chip memory bandwidth is 24 GB/s unless otherwise specified.

TABLE 4: Testing cases for each tensor network structure.

Name	Operations or Algorithms	Kernels
$Op1$	$W^T X$	SpLrMM
$Op2$	$(W^T \odot W^T)\Omega$	SpLrMM
$Op3$	$\Omega_X \circ (WH)$	LrSampling
$Alg1$	Masked Square Loss (ALS or Multiplicative Update)	Fused SpLrMMs
$Alg2$	Masked Square Loss (Gradient Descent)	Fused SpLrMM & LrSampling
$Alg3$	KL Divergence (Multiplicative Update)	Fused SpLrMM & LrSampling

Testing Benchmarks. We totally consider 19 cases to demonstrate our STE's wide support for SpTD algorithms. As mentioned earlier, the algorithm diversity is mainly reflected from three aspects: i) the network structure; ii) the loss function; iii) the iterative method. For the first aspect, we consider the most commonly used structures, i.e. CP, TT, and Tucker; for the last two aspects, we also consider the widely used ones (see Table 4). Note that for Tucker, there is an extra variant $Op1(/c)$ for $Op1$, which is a modified SpLrMM kernel with its core factor tensor excluded and is actually equivalent to SpTTMc.

Unless otherwise specified, the sparse tensor data used in our simulation are randomly generated because we want to sweep the sparsity spectrum in a regular stride to reveal more insights. Besides, we also evaluate the performance over CPU/GPU using some real-world tensor data. Notice that the rank values are set to 16 throughout this section for simplicity.

TABLE 5: Bounding factors and solution suggestions.

	Bounding Factor	Solution Suggestions
(Comp.)	Compute unit	Increase the compute resources
(NoC)	NoC	Decrease PE group size, use compute transfer, or increase NoC channel width
(Mem.1)	Load sparse entries (i.e. $(\mathcal{X}_{ijk}, i, j, k)$)	Increase off-chip memory bandwidth
(Mem.2)	Load factor tensor data (e.g. $A(i,:)$,$B(j,:,:)$)	Increase PE group size, or increase on-chip memory capacity

6.2 Overall Performance

Figure 12 shows the overall performance in FLOPs for different SpTD algorithms and different input data density ($10^{-1} \sim 10^{-9}$).

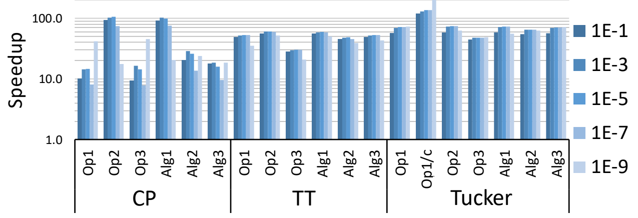


Fig. 13: Speedup over CPU.

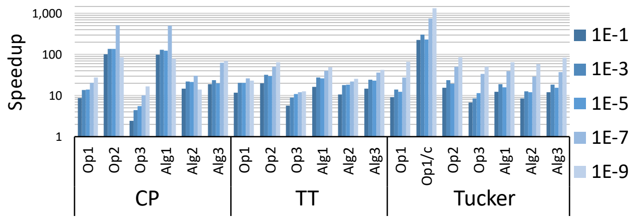


Fig. 14: Speedup over GPU.

The overall performance can reach 82 GFLOPs to 3.9 TFLOPs across a wide range of algorithms and sparsity. We note that the CP family generally presents lower throughput because they usually have less computation per sparse entry compared to the TT and Tucker families. Hence, CP-based algorithms are bounded by loading sparse entries from off-chip memory rather than performing computation as in TT-/Tucker-based algorithms.

In fact, we totally identify four types of factors that might bound the system performance, as listed in Table 5. The performance can be bounded by the compute unit (Comp.), NoC, and loading sparse entries (Mem.1) or factor tensor data (Mem.2) from off-chip memory. Which factor will bound the performance relies on the selection of algorithm, input tensor sparsity and the system configurations such as PE group size and off-chip memory bandwidth.

6.3 Comparison with CPU and GPU

To compare STE with CPU/GPU implementations, we use Taco [47] to generate the corresponding codes for $Op1$, $Op2$, and $Op3$. Because Taco does not support the kernel fusion, the cases $Alg1$, $Alg2$, and $Alg3$ can only be implemented by calling $Op1$, $Op2$, and $Op3$ individually. Although there are other hand-tuned high-performance routines for SpTD on both CPU (e.g. SPLATT [30]) and GPU (e.g. B. Liu et al. [35]), they only support one or two specific kernels that are insufficient for a comprehensive

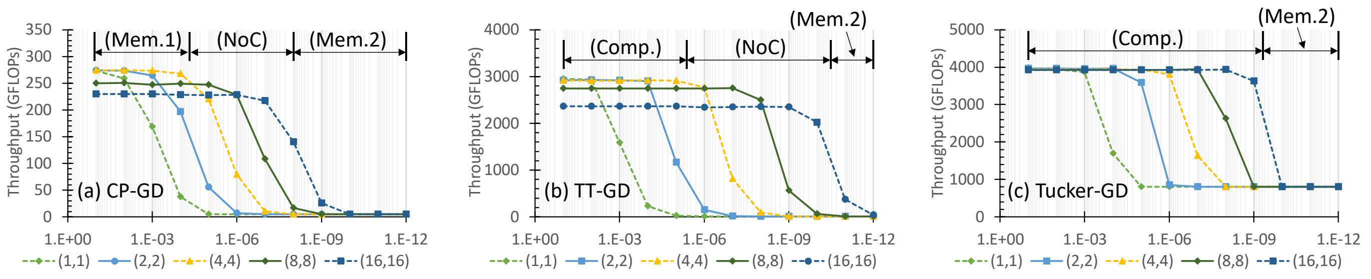


Fig. 15: Performance curve with increasing input tensor sparsity and PE group size.

comparison. On the other hand, the codes generated by Taco provide comparable performance to those of hand-tuned kernels [47]. The CPU platform is 2× Intel(R) Xeon(R) CPU E5-2620 v4 with four DDR4 2133 MHz memories, and we use 32 threads during execution; the GPU platform is NVIDIA Titan V. Note that the default off-chip memory bandwidth of STE is only 24 GB/s, while the same bandwidth specification of CPU and GPU can reach 96 GB/s and >500 GB/s, respectively. To make the comparison fair, we increase the off-chip memory bandwidth of STE to 96 GB/s in this subsection.

Figure 13 and Figure 14 show the speedup over CPU and GPU on synthetic data, respectively. On average, STE can achieve 45× speedup over CPU and 29× speedup over GPU. We observe a better acceleration in CP-$Op2$, CP-$Alg1$, and Tucker-$Op1$(/c), because the performance of these three cases almost solely depends on the factor of "Mem.2", which highlights the unique advantage of STE. In addition, Table 6 shows the speedup over CPU/GPU on real-world data, which shows similar results like that on synthetic data. The data sources are as follows: DNN is a reshaped weight matrix of a pruned fully-connected layer in deep neural networks; Nell2 [48] is a natural language dataset; NIPS [49] is a publication statistic tensor in the Annual Conference on Neural Information Processing Systems.; and Email [50] is a tensor recording email sending/receiving statistics. This table also indicates the general impact of our work in many fields.

TABLE 6: Speedup over CPU and GPU using real-world data.

Name	Alg	Size	Density	STE/CPU	STE/GPU
DNN	CP-Op1	512×512×392	3.0E-01	3.3×	3.4×
Nell2	CP-Op1	12K×9K×28K	2.4E-05	11.1×	14.2×
DNN	Tucker-Op1	512×512×392	3.0E-01	50.1×	7.0×
Nell2	Tucker-Op1	12K×9K×28K	2.4E-05	70.6×	14.4×
NIPS	TT-Op1	2K×3K×14K×17	1.8E-06	40.2×	24.1×
Email	TT-Op1	6K×6K×224K×1K	5.5E-09	33.2×	20.0×

6.4 Tensor Sparsity and PE Group Analysis

In the following analysis, we only select $Alg2$ (GD) to shrink the experiment space. Figure 15 depicts the performance under different input tensor density and PE group size. We observe that the performance degrades with a "ladder-shape" curve as the sparsity increases, which is also observed in other algorithms (see Figure 12). The three distinct regions of the "ladder-shape" curves seem caused by different bounding factors as the sparsity increases. To verify this hypothesis, we measure the resource utilization of different hardware components in Figure 16 and mark our interpretation in Figure 15. Apparently, the loading of factor tensors (Mem.2) will bound the performance in the extremely sparse cases, due to the higher requirement for buffer capacity that exceeds the limit of a PE group. A larger PE group size offers a larger total buffer capacity, thus delaying the turning point of performance descent. However, it is not cost-free: a larger PE group size increases the inter-PE communication distance and thus degrading the performance when the NoC itself becomes the bounding factor. This can be seen from the regions marked with NoC in Figure 15. In this sense, there is not an always correct configuration of PE group size to fit all sparsity levels. We suggest selecting the PE group size according to the actual sparsity of input tensors.

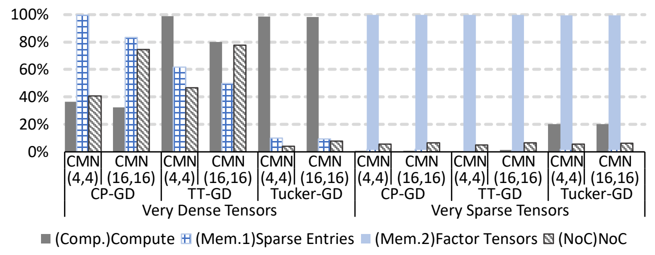


Fig. 16: Resource utilization breakdown.

TABLE 7: Computation and memory costs for each sparse entry. Mem.2 can be amortized due to the inter-sparse entry reuse.

Algorithm	Comp. (MACs)	NoC (Bytes)	Mem.1 (Bytes)	Mem.2 (Bytes)
CP-GD	161	768	14	768
TT-GD	3169	1792	16	8704
Tucker-GD	26225	768	14	66304

We have three more interesting observations from Figure 15. First, in the cases of low sparsity, CP-GD is bounded by Mem.1 (loading sparse entries), while TT-GD and Tucker-GD are bounded by Comp. This is because TT-GD and Tucker-GD have more computations to deal with than CP-GD for each sparse entry (see Table 7). Second, Tucker-GD does not have the NoC bound region. This is due to its significantly higher MACs/traffic ratio (also see Table 7). Third, the throughput of Tucker-GD does not converge to zero even if under very high sparsity. Recalling Figure 2, there is a special core tensor in Tucker decomposition. The core can be reused by all sparse entries, which maintains the least amount of computations.

6.5 Memory Optimization Analysis

In order to study the influence of off-chip memory bandwidth, we measure the performance improvements if we increase the off-chip memory bandwidth to 2×, 3×, and 4× over the 24 GB/s baseline. The results are visualized in Figure 17. The improvements are tightly related to the bounding factors listed in Table 5 and need to be understood with the aid of Figure 15. For CP-GD with dense tensors ($\rho = 10^{-1}$ and $\rho = 10^{-5}$), there are initial performance improvements but they are saturated at $2\times$ bandwidth. The improvement is owing to the initial Mem.1 bound, while the saturation is caused by the latter NoC bound. For CP-GD with sparse tensors ($\rho = 10^{-9}$), the performance improvement is not saturated in our tests because it is always bounded by Mem.2. For TT-GD and Tucker-GD, there is no performance improvement under both $\rho = 10^{-1}$ and $\rho = 10^{-5}$ due to the compute rather than memory bound.

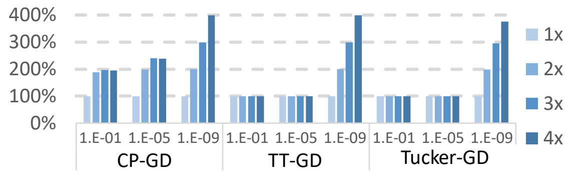


Fig. 17: Normalized performance with increased off-chip memory bandwidth. Here the PE group size is set to 4×4.

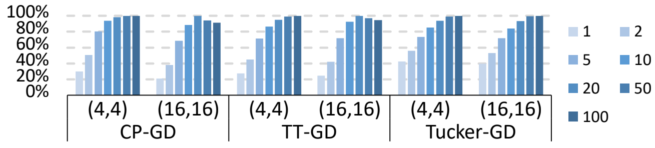


Fig. 18: Normalized performance with increased *SlotMax*. Here the tensor density is 10^{-1}.

Figure 18 depicts the impact of *SlotMax*. A smaller *SlotMax* means less on-the-fly tasks, making the compute unit prone to be idle. We observe an obvious performance penalty under $SlotMax < 20$. When the *SlotMax* comes larger than 50, the performance improvement stops. Therefore, we adopt $SlotMax = 50$ in our architecture design.

6.6 Compute Optimization Analysis

We evaluate the contribution of kernel fusion or compute transfer in Figure 19. Without the kernel fusion optimization, the kernels need to be executed individually, causing redundant computations and off-chip accesses. Note that $Op1$, $Op2$, and $Op3$ cannot achieve benefits from kernel fusion since they are single kernels. Without the compute transfer optimization, i.e. the computation is only allowed to be processed in the home PE and every byte of data on remote PEs must be acquired through the NoC, the performance significantly decreases due to the overwhelmed NoC, sometimes even down to only 8% of the baseline. CP-$Op1$ and Tucker-$Alg3$ do not suffer from the removal of computer transfer because the NoC traffic is not the bound there.

7 Related Works

Algorithms and Applications. Tensor decomposition is a big family rather than a single algorithm. A lot of variants have been proposed , using different tensor network structure (CP [20], Tucker [21], tensor train (TT) [22], or hierarchical-Tucker (HT) [23]), different loss function, different iterative method [51], and sometimes different constraint [52]. The application domains for tensor decomposition are widely identified, such as to reduce data

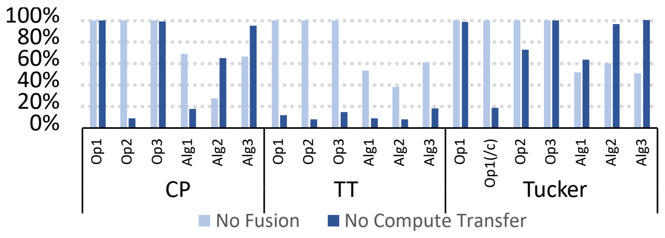


Fig. 19: Normalized performance affected by kernel fusion or compute transfer. Here the tensor density and PE group size are 10^{-5} and 8×8, respectively.

acquisition time in MRI [13], medical and health data analysis [10], [11], [12], solving high-dimensional equations [53], visual data completion [15], EDA problems [54], chatroom modeling model [4], [5], social networks analysis [2], [3], and abnormal network traffic detection [7], [8], [9]. Some works make special assumptions on the data source of tensors, such as Boolean tensor [55], streaming tensor [16], [17], tensor with side information [33], or controllable tensor allowing importance sampling [34], while in this work we study the common cases with sparse input tensors.

Software Optimization. Substantial efforts have been made to optimize the tensor decomposition performance, but there are different directions. S. Smith et al. [31], [38] propose a compact data format for sparse tensor storage. O. Kaya et al. [28] and S. Smith [29] et al. explore different methods to partition the sparse entries into groups. SPLATT [30] is specially tailored to execute MTTKRP on CPUs. Other works have turned into GPUs [35], [36] and distributed platforms [24], [24], [26], [37], [27]. There are also works that compare the performance of different implementations [56] as well as the convergence speed of different optimization algorithms [32]. F. Kjolstad et al. [47] build a code generator, Taco, to express arbitrary contraction, which is also used as our baseline. The above software optimizations on general-purpose processors still do not solve the challenges mentioned in Section 3.

Specialized Hardware. ExTensor [57] is a specialized hardware focusing on the same problem of Taco [47]: handling the complex zero-skipping logic when two or more tensors are sparse. In contrast, SpTD only requires the input tensor to be sparse, thus the specialization of ExTensor is of no use in our context. TIE [58] makes use of TT decomposition for deep learning. However, the tensor-based data in TIE are already decomposed before deploying onto hardware while our work handles the tensor decomposition itself.

8 Conclusion

We identify that the challenges in processing sparse tensor decomposition (SpTD) include the algorithm diversity, requirements for buffer flexibility and data bandwidth due to variable sparsity, and the hardness to fuse kernels. To address these issues, we propose a unified abstraction via two general sparse kernels (i.e. SpLrMM and SpSampling) and a unified execution framework that can accommodate most algorithms with kernel fusion. Then, we design a specialized accelerator, STE, to implement our top-down solution. The PE-interactive architecture enables the sharing of local memory capacity/bandwidth of each PE and avoids write conflicts during kernel fusion. The deadlock is also eliminated by identifying and satisfying three requirements during task mapping. Through extensive experiments, we demonstrate an average speedup of $45\times$

over CPU and 29× over GPU. Comprehensive analysis of the impact of tensor sparsity, PE group size, and memory/compute optimizations are further presented to give design guidance. Our design can stimulate more researches in designing specialized architectures for high-performance tensor decomposition.

REFERENCES

[1] B. W. Bader, T. G. Kolda, and R. A. Harshman, "Temporal analysis of social networks using three-way dedicom." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); Sandia . . . , Tech. Rep., 2006.

[2] S. Sizov, S. Staab, and T. Franz, *Analysis of Social Networks by Tensor Decomposition*. Boston, MA: Springer US, 2010, pp. 45–58. [Online]. Available: https://doi.org/10.1007/978-1-4419-7142-5_3

[3] A. Kao, W. Ferng, S. Poteet, L. Quach, and R. Tjoelker, "Talison - tensor analysis of social media data," in *2013 IEEE International Conference on Intelligence and Security Informatics*, June 2013, pp. 137–142.

[4] E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener, "Modeling and multiway analysis of chatroom tensors," in *Intelligence and Security Informatics*, P. Kantor, G. Muresan, F. Roberts, D. D. Zeng, F.-Y. Wang, H. Chen, and R. C. Merkle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 256–268.

[5] E. Acar, S. A. Çamtepe, and B. Yener, "Collective sampling and analysis of high order tensors for chatroom communications," in *Intelligence and Security Informatics*, S. Mehrotra, D. D. Zeng, H. Chen, B. Thuraisingham, and F.-Y. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 213–224.

[6] M. M. Baskaran, T. Henretty, J. Ezick, R. Lethin, and D. Bruns-Smith, "Enhancing network visibility and security through tensor analysis," *Future Generations Computer Systems*, 2 2019.

[7] K. Xie, L. Wang, X. Wang, G. Xie, J. Wen, and G. Zhang, "Accurate recovery of internet traffic data: A tensor completion approach," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[8] K. Xie, L. Wang, X. Wang, G. Xie, J. Wen, G. Zhang, J. Cao, and D. Zhang, "Accurate recovery of internet traffic data: A sequential tensor completion approach," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 793–806, April 2018.

[9] K. Xie, C. Peng, X. Wang, G. Xie, J. Wen, J. Cao, D. Zhang, and Z. Qin, "Accurate recovery of internet traffic data under variable rate measurements," *IEEE/ACM Transactions on Networking*, vol. 26, no. 3, pp. 1137–1150, June 2018.

[10] I. Perros, R. Chen, R. Vuduc, and J. Sun, "Sparse hierarchical tucker factorization and its application to healthcare," in , 11 2015, pp. 943–948.

[11] J. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 08 2014.

[12] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. N. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *KDD*, 2015.

[13] J. He, Q. Liu, A. Christodoulou, C. Ma, F. Lam, and Z.-P. Liang, "Accelerated high-dimensional mr imaging with sparse sampling using low-rank tensors," *IEEE Transactions on Medical Imaging*, vol. 35, pp. 2119–2129, 04 2016.

[14] S. Gandy, B. Recht, and I. Yamada, "Tensor completion and low-n-rank tensor recovery via convex optimization," *Inverse Problems*, vol. 27, no. 2, p. 025010, jan 2011. [Online]. Available: https://doi.org/10.1088%2F0266-5611%2F27%2F2%2F025010

[15] J. Liu, P. Musialski, P. Wonka, and J. Ye, "Tensor completion for estimating missing values in visual data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 208–220, Jan 2013.

[16] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis, *Streaming Tensor Factorization for Infinite Data Sources*. , , pp. 81–89. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611975321.10

[17] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: Dynamic tensor analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 374–383. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150445

[18] C. Hawkins and Z. Zhang, "Robust factorization and completion of streaming tensor data via variational bayesian inference," in , 2018.

[19] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. Phan, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE Signal Processing Magazine*, vol. 32, no. 2, pp. 145–163, 2015.

[20] J. D. Caroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via n-way generalization of eckart-young decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

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

[22] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[23] L. Grasedyck, "Hierarchical singular value decomposition of tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 4, p. 2029–2054, 2010.

[24] N. Park, B. Jeon, J. Lee, and U. Kang, "Bigtensor: Mining billion-scale tensor made easy," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: ACM, 2016, pp. 2457–2460. [Online]. Available: http://doi.acm.org/10.1145/2983323.2983332

[25] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. New York, NY, USA: ACM, 2012, pp. 316–324. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339583

[26] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *2015 IEEE 31st International Conference on Data Engineering*, April 2015, pp. 1047–1058.

[27] S. Smith and G. Karypis, "Dms : Distributed sparse tensor factorization with alternating least squares," in , 2015.

[28] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–11.

[29] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 902–911.

[30] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 61–70.

[31] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:7. [Online]. Available: http://doi.acm.org/10.1145/2833179.2833183

[32] S. Smith, J. Park, and G. Karypis, "An exploration of optimization algorithms for high performance tensor completion," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 359–371.

[33] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing, "Flexifact: Scalable flexible factorization of coupled tensors on hadoop," in *Proceedings of the 2014 SIAM International Conference on Data Mining*. SIAM, 2014, pp. 109–117.

[34] E. Papalexakis, C. Faloutsos, and N. Sidiropoulos, "Parcube: Sparse parallelizable candecomp-parafac tensor decomposition," *ACM Transactions on Knowledge Discovery from Data*, vol. 10, no. 1, 7 2015.

[35] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 47–57.

[36] J. Li, Y. Ma, and R. Vuduc, "ParTI! : A parallel tensor infrastructure for multicore cpus and gpus," Oct 2018. [Online]. Available: https://github.com/hpcgarage/ParTI

[37] J. H. Choi and S. V. N. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 1296–1304. [Online]. Available: http://dl.acm.org/citation.cfm?id=2968826.2968971

[38] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 653–668.

[39] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, "Optimizing sparse tensor times matrix on gpus," *Journal of Parallel and Distributed Computing*, vol. 129, pp. 99 – 109, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518305161

[40] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, P. Murali, S. S. Pandian, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for sparse tensors," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New

York, NY, USA: ACM, 2018, pp. 374–384. [Online]. Available: http://doi.acm.org/10.1145/3205289.3205315

[41] O. Kaya and B. Uçar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 103–112.

[42] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[43] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 01 2009.

[44] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, May 2012, pp. 201–210.

[45] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[46] "Berkeley fpu," https://github.com/ucb-bar/berkeley-hardfloat, accessed: 2019-09-30.

[47] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, "Taco: A tool to generate tensor algebra kernels," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 943–948.

[48] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning." in *AAAI*, vol. 5, 2010, p. 3.

[49] A. Globerson, G. Chechik, F. Pereira, and N. Tishby, "Euclidean Embedding of Co-occurrence Data," *The Journal of Machine Learning Research*, vol. 8, pp. 2265–2295, 2007.

[50] J. Shetty and J. Adibi, "The enron email dataset database schema and brief statistical report," *Information sciences institute technical report, University of Southern California*, vol. 4, 2004.

[51] T. Maehara, K. Hayashi, and K.-i. Kawarabayashi, "Expected tensor decomposition with stochastic gradient descent," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 1919–1925. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016100.3016167

[52] S. Smith, A. Beri, and G. Karypis, "Constrained tensor factorization with accelerated ao-admm," in *2017 46th International Conference on Parallel Processing (ICPP)*, Aug 2017, pp. 111–120.

[53] L. Grasedyck, D. Kressner, and C. Tobler, "A literature survey of low-rank tensor approximation techniques," *GAMM-Mitteilungen*, vol. 36, no. 1, pp. 53–78, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/gamm.201310004

[54] Z. Zhang, K. Batselier, H. Liu, L. Daniel, and N. Wong, "Tensor computation: A new framework for high-dimensional problems in eda," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 521–536, 2016.

[55] N. Park, S. Oh, and U. Kang, "Fast and scalable distributed boolean tensor factorization," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, April 2017, pp. 1071–1082.

[56] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Performance evaluation of parallel sparse tensor decomposition implementations," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 54–57. [Online]. Available: https://doi.org/10.1109/IA3.2016.14

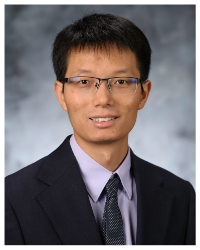
[57] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 319–333.

[58] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "Tie: energy-efficient tensor train-based inference engine for deep neural network," in *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 2019, pp. 264–278.

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

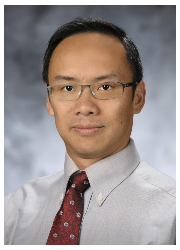
Lei Deng received the B.E. degree from University of Science and Technology of China, Hefei, China in 2012, and the Ph.D. degree from Tsinghua University, Beijing, China in 2017. He is currently a Postdoctoral Fellow at the Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, USA. His research interests span the area of brain-inspired computing, machine learning, neuromorphic chip, computer architecture, tensor analysis, and complex networks. Dr. Deng has authored or co-authored over 40 refereed publications. He was a PC member for *ISNN* 2019. He currently serves as a Guest Associate Editor for *Frontiers in Neuroscience* and *Frontiers in Computational Neuroscience*, and a reviewer for a number of journals and conferences. He was a recipient of MIT Technology Review Innovators Under 35 China 2019.

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



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

He received the Best Paper Award for the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* in 2014, the Best Paper Award for the *IEEE Transactions on Components, Packaging and Manufacturing Technology* in 2018, two Best Paper Awards (*IEEE EPEPS* 2018 and *IEEE SPI* 2016), and three additional Best Paper Nominations (*CICC* 2014, *ICCAD* 2011, and *ASP-DAC* 2011) at international conferences. His Ph.D. dissertation was recognized by the ACM SIGDA Outstanding Ph.D. Dissertation Award in Electronic Design Automation in 2016, and by the Doctoral Dissertation Seminar Award (i.e., Best Thesis Award) from the Microsystems Technology Laboratory, MIT in 2015. He was a recipient of the Li Ka-Shing Prize from the University of Hong Kong in 2011.



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

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