Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations

Nitish Srivastava[1*], Hanchen Jin[1], Shaden Smith[2], Hongbo Rong[3],
David Albonesi[1*], Zhiru Zhang[1*]
[1]Cornell University
[2]Microsoft AI & Research
[3]Intel Parallel Computing Lab
*{nks45, dha7, zhiruz}@cornell.edu

ABSTRACT

Tensor factorizations are powerful tools in many machine learning and data analytics applications. Tensors are often sparse, which makes sparse tensor factorizations memory bound. In this work, we propose a hardware accelerator that can accelerate both dense and sparse tensor factorizations. We co-design the hardware and a sparse storage format, which allows accessing the sparse data in vectorized and streaming fashion and maximizes the utilization of the memory bandwidth. We extract a common computation pattern that is found in numerous matrix and tensor operations and implement it in the hardware. By designing the hardware based on this common compute pattern, we can not only accelerate tensor factorizations but also mixed sparse-dense matrix operations. We show significant speedup and energy benefit over the state-of-the-art CPU and GPU implementations of tensor factorizations and over CPU, GPU and accelerators for matrix operations.

1. INTRODUCTION

Tensor algebra lives at the heart of machine learning (ML). Classical ML techniques such as embedding generation in recommender systems, dimensionality reduction, latent Dirichlet allocation, compression on neural networks and detecting cliques in social networks make use of tensor factorizations [1–4]. Tensor factorizations have traditionally been used in recommender systems [5, 6] to produce factor matrices that represent an embedding into the reduced latent space. While deep neural networks are expensive to train, require a large number of labeled data and have limited interpretability, tensor factorizations provide a faster, more interpretable, yet competitive method for producing embedding for recommender systems [7]. Recently, tensor factorizations have also achieved promising results in compressing deep neural networks [8–10].

There are two popular methods for tensor factorizations [11]: canonical polyadic decomposition (CPD) and Tucker decomposition. CPD approximates the tensor as a sum of rank-one tensors, whereas Tucker approximates it by a core tensor (weights) and factor matrices (principal components) along each mode [11]. The two most expensive computation kernels for these factorizations are matricized tensor times Khatri Rao product (MTTKRP) and tensor times matrix chain (TTMc).

Traditionally, tensor factorizations have been done on CPUs and GPUs, both of which have low energy efficiency as they allocate excessive hardware resources to flexibly support various workloads [12–14]. Hardware specialization has become a common means to improve the compute efficiency. However, there are two key challenges with designing an accelerator for tensor factorizations. First, many of the real-world tensors such as Netflix movie ratings [15] and never-ending language learning (NELL) [16] are sparse, which makes tensor factorizations memory bound. Second, the compute and memory access patterns of different tensor factorizations can be very different, which makes it necessary to reduce these computations into some basic operations and implement them in the hardware.

In this work, we propose *Tensaurus*, a new hardware accelerator for highly efficient processing of mixed sparse and dense tensor computations. Tensaurus accelerates a computation pattern that we observe in common across different tensor factorization kernels as well as several widely used matrix operations. Our accelerator further exploits a new sparse storage format that we introduce to allow accessing of sparse data in a vectorized and streaming manner to achieve high memory bandwidth utilization. Thus, with the co-design of the accelerator architecture and sparse storage, Tensaurus is both *versatile* and *adaptable*. It is versatile in the sense that it can accelerate both tensor factorizations and common matrix operations such as matrix-matrix and matrix-vector multiplications, which are key compute primitives in many ML applications. It can also easily adapt to different levels of sparsity found in these computations.

To the best of our knowledge, no prior work has proposed a hardware accelerator for sparse tensor factorizations and previous efforts have been focusing on dense tensor factorizations (e.g., T2S-Tensor [17] and [18]).

The key technical contributions of this paper are:

1. We are the first to propose a hardware accelerator that is capable of accelerating not only sparse tensor factorizations, but also dense tensor factorizations and other common mixed sparse-dense (sparse-dense and dense-dense) matrix operations for a wide range of sparsity.

2. We introduce a new sparse storage format, *compressed interleaved sparse slice* (*CISS*), which allows accessing sparse data in a vectorized and streaming manner and thus achieves high memory bandwidth utilization and performance for sparse tensor kernels.

3. We achieve significant speedup and energy reduction for tensor factorizations over the state-of-the-art CPU and GPU implementations. We also compare our accelerator against CPU, GPU, and the state-of-the-art hardware accelerator for sparse CNNs and demonstrate higher performance and energy efficiency.

2. BACKGROUND

2.1 Tensor Notations

A tensor is a generalization of a matrix to multiple dimensions. A scalar is a tensor of dimension zero, a vector is a tensor of dimension one and a matrix is a tensor of dimension two. We denote tensors with three or more dimensions using capital calligraphic letters (e.g., $\boldsymbol{\mathcal{A}}$), matrices using boldface capital letters (e.g., $\mathbf{A}$), vectors using boldface letters (e.g., $\mathbf{a}$), and scalars using Greek letters (e.g. α).

The dimensions of a tensor are also called its modes and colon(:) is used to indicate all the elements of a mode. Thus a 3-dimensional (3-d) tensor is a tensor with 3 modes. Fig. 3a shows an example of a $4\times2\times2$ 3-d tensor where i, j and k are the mode 0, mode 1 and mode 2 indices of the data elements. Fibers are building blocks of tensors. A fiber is the result of holding all but one index constant. For a 3-d tensor $\boldsymbol{\mathcal{A}}$, its fibers are $\boldsymbol{\mathcal{A}}(:,j,k)$, $\boldsymbol{\mathcal{A}}(i,:,k)$, and $\boldsymbol{\mathcal{A}}(i,j,:)$. Similarly, for a matrix $\mathbf{A}$ its rows $\mathbf{A}(i,:)$ and columns $\mathbf{A}(:,j)$ are its fibers. A slice of a tensor is the resultant matrix by holding all but two indices constant. Slices of a 3-d tensor $\boldsymbol{\mathcal{A}}$ would be $\boldsymbol{\mathcal{A}}(i,:,:)$, $\boldsymbol{\mathcal{A}}(:,j,:)$ and $\boldsymbol{\mathcal{A}}(:,:,k)$.

2.2 MTTKRP

MTTKRP is the key computation kernel in the alternating least square (ALS) method, which is the most popular method for finding the factor matrices in CPD [1,11]. The computation for MTTKRP consists of multiplication of a tensor with $N-1$ factor matrices, where N is the mode of the tensor, to produce an output matrix. Eq. (1) and Fig. 1 show the MTTKRP kernel for a 3-d tensor along mode 0 (i), where $\cdot$ denotes multiplication. Since MTTKRP is used for both sparse and dense tensor factorizations [19], we refer to MTTKRP on sparse tensors as SpMTTKRP and on dense tensors as DMTTKRP. The operand matrices and the output matrix in both SpMTTKRP and DMTTKRP are dense. Even with very efficient data structures [20,21], the arithmetic intensity of SpMTTKRP remains low, making this kernel memory bound [22].

The Hadamard product, denoted by $\circ$, is the element-wise multiplication of two matrices with the same size. It is distributive and can be used to factor out the operand matrices in MTTKRP [20] as shown in Eq. (2). Here the Hadamard product operates on two vectors instead of two matrices. Such factorization reduces the number of multiplications in DMTTKRP from $2\cdot I\cdot J\cdot K\cdot F$ to $I\cdot J\cdot F\cdot(K+1)$. Here I, J and K are the sizes of the three dimensions of the tensor and F is the desired rank for tensor factorization (normally in the order of 10s or 100s). Similar reductions are observed in the case of SpMTTKRP [20]. Eq. (2) can be easily generalized to MTTKRP on tensors with more than three dimensions as shown in Eq. (3).

$$\mathbf{Y}(i,f)=\sum_{j=0}^{J-1}\sum_{k=0}^{K-1}\boldsymbol{\mathcal{A}}(i,j,k)\cdot\mathbf{B}(j,f)\cdot\mathbf{C}(k,f) \tag{1}$$

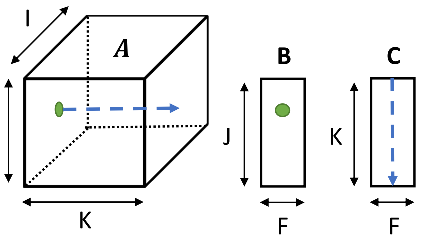


Figure 1: **MTTKRP**

$$\mathbf{Y}(i,:)=\sum_{j=0}^{J-1}\mathbf{B}(j,:)\circ\left(\sum_{k=0}^{K-1}\boldsymbol{\mathcal{A}}(i,j,k)\cdot\mathbf{C}(k,:)\right) \tag{2}$$

$$\mathbf{Y}(i_1,:)=\sum_{i_2}\mathbf{M}_2(i_2,:)\circ\ldots\circ\sum_{i_n}\boldsymbol{\mathcal{A}}(i_1,\ldots,i_n)\cdot\mathbf{M}_n(i_n,:) \tag{3}$$

2.3 TTMc

TTMc is the key computation kernel in higher-order orthogonal iterations (HOOI) [23], which is the most popular method for finding the core tensor and factor matrices in Tucker decomposition [1,11]. TTMc involves a sequence of tensor times matrix operations, which compresses the tensor. The output of TTMc is another tensor compressed for all but one mode. Eq. (4) shows the TTMc kernel for a 3-d tensor along mode 0 (i). Similar to MTTKRP, TTMc is used for both dense and sparse tensors [24–26]. For now, we refer to TTMc on sparse tensors as SpTTMc and on dense tensors as DTTMc. For both SpTTMc and DTTMc, the operand matrices and output tensor are dense.

The Kronecker product, denoted by $\otimes$, is the generalization of vector outer product to matrices and tensors. It is also distributive and can be used to factor out the operand matrices in TTMc as shown in Eq. (5). Such factorization reduces the number of multiplications in DTTMc from $2\cdot I\cdot J\cdot K\cdot F_1\cdot F_2$ to $I\cdot J\cdot(KF_2+F_1F_2)$ and similar reductions are observed in the case of SpTTMc [27]. Here F_1 and F_2 are the desired ranks for tensor factorization and are on the order of 10s or 100s. Eq. (5) can also be easily generalized to tensors with more than three dimensions as shown in Eq. (6).

$$\boldsymbol{\mathcal{Y}}(i,f_1,f_2)=\sum_{j=0}^{J-1}\sum_{k=0}^{K-1}\boldsymbol{\mathcal{A}}(i,j,k)\cdot\mathbf{B}(j,f_1)\cdot\mathbf{C}(k,f_2) \tag{4}$$

$$\boldsymbol{\mathcal{Y}}(i,:,:)=\sum_{j=0}^{J-1}\mathbf{B}(j,:)\otimes\left(\sum_{k=0}^{K-1}\boldsymbol{\mathcal{A}}(i,j,k)\cdot\mathbf{C}(k,:)\right) \tag{5}$$

$$\boldsymbol{\mathcal{Y}}(i_1,:,\ldots,:)=\sum_{i_2}\mathbf{M}_2(i_2,:)\otimes\ldots\otimes\sum_{i_n}\boldsymbol{\mathcal{A}}(i_1,\ldots,i_n)\cdot\mathbf{M}_n(i_n,:) \tag{6}$$

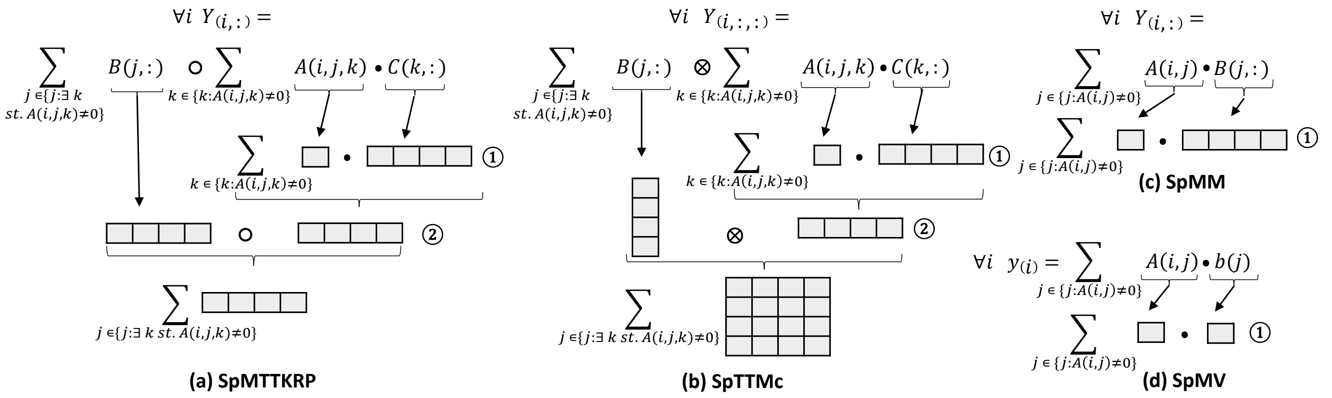


Figure 2: **SpMTTKRP, SpTTMc, SpMM, and SpMV expressed using the SF^3 compute pattern in Eq.** (9).

2.4 Matrix-Matrix Multiplication

Matrix-matrix multiplication involves multiplication of two matrices to produce an output matrix as shown in Eq. (7). Matrix-matrix multiplication involving two dense matrices is known as GEMM, and a sparse matrix and a dense matrix is known as SpMM. GEMM and SpMM are building blocks of many algorithms such as graph learning [28, 29] and CNNs [30].

$$\mathbf{Y}(i,:)=\sum_{j=0}^{J-1}\mathbf{A}(i,j)\cdot\mathbf{B}(j,:) \tag{7}$$

2.5 Matrix-Vector Multiplication

Matrix-vector multiplication involves multiplication of a matrix with a vector to produce an output vector as shown in Eq. (8). Matrix-vector multiplication involving a dense matrix and a dense vector is known as GEMV, and a sparse matrix and a dense vector is known as SpMV. GEMV and SpMV are used in applications such as PageRank [31], RNNs, minimal spanning tree, single-source shortest path and ML algorithms such as support vector machine [32] and text analytics [33].

$$\mathbf{y}(i)=\sum_{j=0}^{J-1}\mathbf{A}(i,j)\cdot\mathbf{b}(j) \tag{8}$$

3. COMPUTE PATTERN

We observe that a common compute pattern can be extracted across all the aforementioned kernels, namely, MTTKRP, TTMc, matrix-matrix multiplication, and matrix-vector multiplication. We name this compute pattern as *scalar-fiber product followed by fiber-fiber products (SF^3)* and it is expressed in the following form:

$$\mathbf{fibers}_{out} = \sum_{D_1}\mathbf{fiber}_1\ op \sum_{D_0}\big(\mathbf{scalar}\cdot\mathbf{fiber}_0\big) \tag{9}$$

Here, $\mathbf{fibers}_{out}$ represent one or more output fibers of a tensor, $\mathbf{fibers}_0$ and $\mathbf{fibers}_1$ represent a single fiber from two tensors, **scalar** represents a scalar value from a tensor, *op* is either a Hadamard product of two vectors ($\circ$) or a Kronecker product of two vectors ($\otimes$), and D_1 and D_0 are domains over which the two summations are performed.

Table 1 shows how different tensor computations map to SF^3 compute pattern. For DMTTKRP, $\mathbf{fibers}_{out}$ is a row from $\mathbf{Y}$ matrix, $\mathbf{fiber}_1$ is a row from $\mathbf{B}$ matrix, *op* is $\circ$, **scalar** is data value from $\boldsymbol{\mathcal{A}}$ tensor and $\mathbf{fiber}_0$ is a row from $\mathbf{C}$ matrix. For DTTMc, all the notations are the same as DMTTKRP, except $\mathbf{fibers}_{out}$ are more than one fiber (one slice $F_1\times F_2$) from $\boldsymbol{\mathcal{Y}}$ tensor and *op* is $\otimes$. For GEMM, the **scalar** is data from $\mathbf{A}$ matrix, $\mathbf{fiber}_0$ is a row from $\mathbf{B}$ matrix, $\mathbf{fibers}_{out}$ is a row from $\mathbf{Y}$ matrix and *op* and $\mathbf{fiber}_1$ are not applicable (NA). GEMV is same as GEMM except $\mathbf{fibers}_{out}$ and $\mathbf{fibers}_0$ consist of only one element. Since all these computations are dense, the domains D_0 and D_1 for each of these computations are also continuous ranges.

Table 1 and Fig. 2 also show how SpMTTKRP, SpTTMc, SpMM, and SpMV map to SF^3 compute pattern. Here the mapping of each sparse kernel to Eq. (9) is the same as that of their dense counterparts except for the domains D_0 and D_1, which are non-continuous ranges and determined by the position of non-zero entries in the sparse tensor. Although shown for 3-d tensors, Eq. (9) can be easily extended to support tensor computations involving tensors with more than three dimensions.

The formulation in Eq. (9) exhibits coarse-grained parallelism, where different output fibers can be computed in parallel and fine-grained single instruction multiple data (SIMD) parallelism where the computation of a single fiber can be performed in a vectorized manner.

4. SPARSE FORMATS

Sparse tensor computations require a sparse storage format that is efficient for both load balancing and parallel data accesses in order to accelerate them on spatial hardware. The existing sparse storage formats such as compressed sparse row (CSR) [34], compressed sparse fiber (CSF) [20], co-ordinate (COO) and their variants [35, 36] allocate data needed by different processing elements (PEs) at far away locations in memory, resulting in low memory bandwidth utilization.

Figs. 3a and 3b show a sparse tensor and how it is stored in an extended CSR format. In this format, all the non-zero entries in the tensor are stored contiguously in the memory along with their mode 1 and mode 2 indices j and k. An array of slice pointers whose length is equal to the number of slices in the tensor (4 in this example) stores the pointers to the beginning of each slice in the array of non-zero entries. Fig. 3c depicts cycle-by-cycle execution of two PEs, each of

Table 1: Mapping of DMTTKRP, SpMTTKRP, DTTMc, SpTTMc, GEMM, SpMM, GEMV, and SpMV kernels to the SF^3 compute pattern in Eq. (9). Here NA means not applicable.

	$\mathbf{fibers}_{out}$	$\mathbf{fiber}_1$	$\mathbf{D}_1$	*op*	**scalar**	$\mathbf{fiber}_0$	$\mathbf{D}_0$
DMTTKRP	$\mathbf{Y}(i,:)$	$\mathbf{B}(j,:)$	$[0,J)$	$\circ$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$[0,K)$
SpMTTKRP	$\mathbf{Y}(i,:)$	$\mathbf{B}(j,:)$	$\{j \mid \exists k\ st.\ \boldsymbol{\mathcal{A}}(i,j,k)\neq 0\}$	$\circ$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$\{k \mid \boldsymbol{\mathcal{A}}(i,j,k)\neq 0\}$
DTTMc	$\boldsymbol{\mathcal{Y}}(i,:,:)$	$\mathbf{B}(j,:)$	$[0,J)$	$\otimes$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$[0,K)$
SpTTMc	$\boldsymbol{\mathcal{Y}}(i,:,:)$	$\mathbf{B}(j,:)$	$\{j \mid \exists k\ st.\ \boldsymbol{\mathcal{A}}(i,j,k)\neq 0\}$	$\otimes$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$\{k \mid \boldsymbol{\mathcal{A}}(i,j,k)\neq 0\}$
GEMM	$\mathbf{Y}(i,:)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{B}(j,:)$	$[0,J)$
SpMM	$\mathbf{Y}(i,:)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{B}(j,:)$	$\{j \mid \mathbf{A}(i,j)\neq 0\}$
GEMV	$\mathbf{y}(i)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{b}(j)$	$[0,J)$
SpMV	$\mathbf{y}(i)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{b}(j)$	$\{j \mid \mathbf{A}(i,j)\neq 0\}$

which reads a different slice of the tensor from the memory in a streaming fashion. As it can be seen in each cycle the two PEs read the data from non-contiguous memory locations.

The *compressed interleaved sparse row* (CISR) format proposed by Fowers *et al.* [37] tackles this issue by storing the sparse data accessed by different PEs at the same time in contiguous memory locations; however, this format requires centralized row decoding, lock-step execution, and applies only to matrices. Partly inspired by CISR, we propose a new sparse storage format called *compressed interleaved sparse slice (CISS).* With this new format, we overcome the limitations of CISR and extend it to tensors with more than two dimensions. Fig. 3d shows the tensor in Fig. 3a stored in the CISS format, which consists of an array of CISS entries. Each entry is $(dw{+}2{\cdot}iw){\cdot}P$ bits long, where dw (data width) and iw (index width) are the bitwidths of the non-zero elements and their mode indices, respectively and P is the number of PEs in hardware. For each PE, a CISS entry consists of three fields: *nnz* (non-zero data value), i/j (mode 0 or mode 1 index) and k (mode 2 index). Since *nnz* is supposed to carry only non-zero data values, a 0 in *nnz* indicates that the i/j field consists of i value and a non-zero in *nnz* indicates that i/j consists of a j value.

To store a sparse tensor in CISS format, first each PE is assigned a slice of the tensor corresponding to its ID. For example, in Fig. 3d PE0 is assigned slice 0 and PE1 is assigned slice 1 in the first cycle. The slice indices for each PE are written to the i/j and the *nnz* is set to 0. In the next few cycles, the CISS entries for a PE are filled with the non-zero entries from the slice by assigning the non-zero data elements to *nnz*, mode 1 indices to i/j and mode 2 indices to k. When all the non-zero elements in the slice assigned to a PE are scheduled, the next available slice is assigned to that PE and its slice index and data values are inserted into the array of CISS entries. For example, in Fig. 3d when all the non-zero entries from the slice $i{=}1$ are inserted into the array, the next available slice is slice $i{=}2$ and hence it gets assigned to PE1.

Since CISS assigns the non-zeros accessed from different PEs at the same time in contiguous memory locations, it achieves high *spatial locality* and *memory bandwidth utilization.* CISS format also schedules the next available slice of the tensor to the PE with the least data that ensures a *load balanced* schedule where each PE is assigned a similar number of non-zero entries. Although described for 3-d tensors, the CISS format can be easily generalized to 2-d matrices and tensors with more than three dimensions.

Fig. 3e also shows the achieved bandwidth when multiple PEs stream a 3-d tensor stored in extended CSR and CISS formats from the off-chip memory with a peak bandwidth of 16 GB/s. As it can be seen, the achieved bandwidth for extended CSR saturates at 1.9 GB/s for 8 PEs while CISS achieves a bandwidth of 11.2 GB/s, very close to the peak bandwidth.

5. TENSAURUS ARCHITECTURE

In this section, using SpMTTKRP as a driving example, we explain the implementation of the SF^3 compute pattern described in Section 3.

Fig. 4 shows the execution of SpMTTKRP kernel using the same tensor as in Fig. 3a on two PEs. The slices $i{=}0$ and $i{=}3$ are assigned to PE0 and slices $i{=}1$ and $i{=}2$ are assigned to PE1 (same as in Fig. 3). Each PE reads a non-zero data element a_{ijk} from the sparse tensor $\boldsymbol{\mathcal{A}}$, the k^{th} row from matrix $\mathbf{C}$ ($\mathbf{c}_{k:}$) and performs a scalar-vector multiplication to produce $\mathbf{t}^k_{ij:}$. All the partial results $\mathbf{t}^k_{ij:}$ for different values of k are accumulated together and then multiplied by the j^{th} row of matrix $\mathbf{B}$ ($\mathbf{b}_{j:}$) to produce the partial results $\mathbf{y}^j_{i:}$. All the $\mathbf{y}^j_{i:}$ vectors are then summed together for different values of j to produce the i^{th} row of the output matrix $\mathbf{Y}$. From this example, it can be seen that the core operations in the SpMTTKRP are scalar-vector multiplication ($a_{ijk}{\cdot}\mathbf{c}_{k:}$), element-wise vector-vector multiplication ($\mathbf{t}^k_{ij:}\circ\mathbf{b}_{j:}$) and element-wise vector-vector addition ($\mathbf{t}^k_{ij:}{+}\mathbf{t}^{k'}_{ij:}$ and $\mathbf{y}^j_{i:}{+}\mathbf{y}^{j'}_{i:}$). There are also two types of intermediate results produced in the SpMTTKRP computation: $\mathbf{t}^k_{ij:}$ and $\mathbf{y}^j_{i:}$.

For the SF^3 compute pattern in Eq. (9), the scalar-vector multiplications arise from $\mathbf{scalar}\cdot\mathbf{fiber}_0$; the element-wise vector-vector multiplication (VVMUL) operations arises from *op*; and the element-wise vector-vector addition (VVADD) arises from the two summations over D_0 and D_1. The intermediate results in Eq. (9) correspond to the partial results of the computations $\mathbf{scalar}{\cdot}\mathbf{fiber}_0$ and $\mathbf{fiber}_1$ $op \sum_{D_0}\mathbf{scalar}{\cdot}\mathbf{fiber}_0$. Thus, using the three major operations: scalar-vector multiplication, VVMUL and VVADD and two sets of storage registers for the two kinds of partial results, we can design the micro-architecture of a PE in Tensaurus. For scalability, we can further split a single vector operation into multiple small vector operations, each of vector length VLEN.

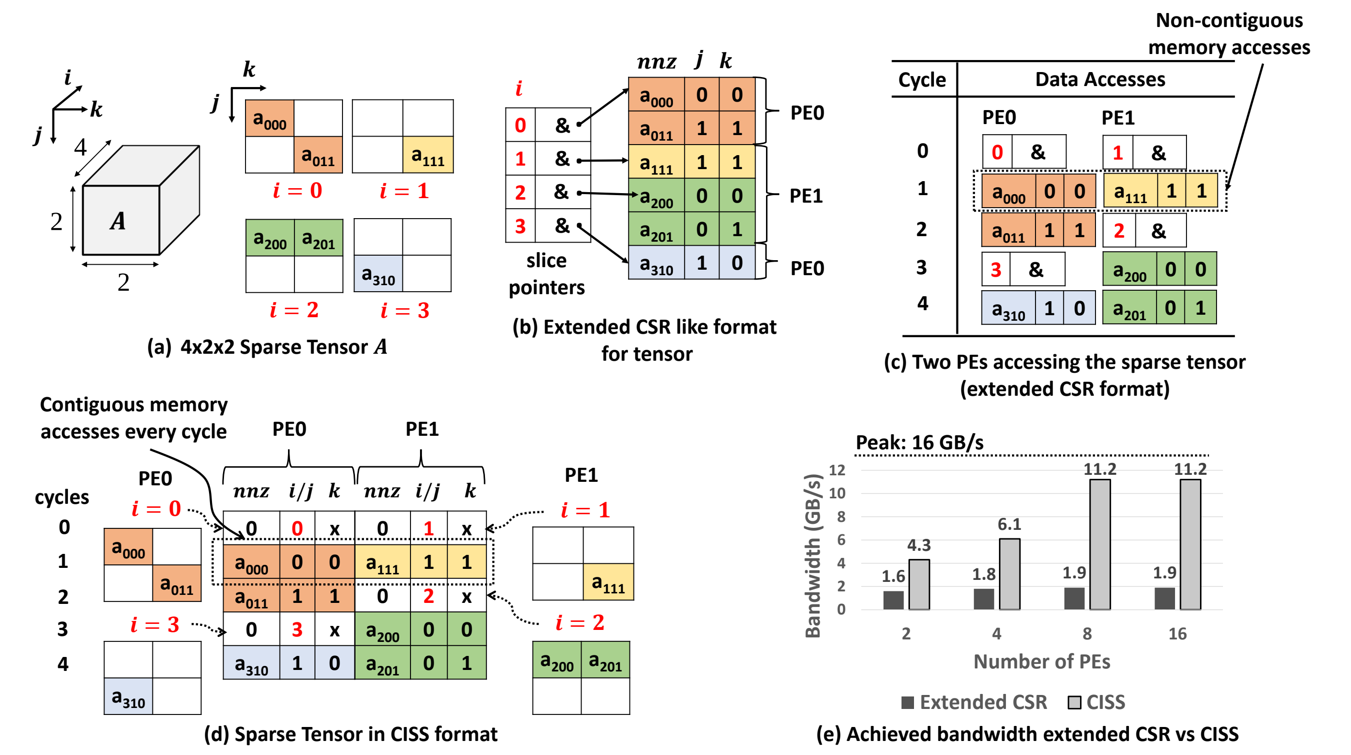

(a) 4x2x2 Sparse Tensor A
(b) Extended CSR like format for tensor
slice pointers
PE0
PE1
(c) Two PEs accessing the sparse tensor (extended CSR format)
Non-contiguous memory accesses
Cycle
Data Accesses
Contiguous memory accesses every cycle
(d) Sparse Tensor in CISS format
Peak: 16 GB/s
Bandwidth (GB/s)
Number of PEs
Extended CSR
CISS
(e) Achieved bandwidth extended CSR vs CISS

Figure 3: **Storage formats** – (a) a 4×2×2 sparse tensor $\mathcal{A}$; (b) the sparse tensor stored in extended CSR format. Here the slice pointers point to beginning of a slice in the array consisting of non-zero data elements and their mode 1 and mode 2 indices j and k. The data in sparse tensor is split across two PEs, where PE0 accesses data from the first slice $i=0$ and $i=3$, and PE1 accesses data from slices $i=1$ and $i=2$. The reference sign **&** inside the white boxes represent pointer values; (c) two PEs accessing the data from sparse tensor stored in extended CSR format shown in (b). Here each PE first reads a slice pointer and then the non-zero values from that slice. The data accessed by the two PEs in the same cycle is stored in non-contiguous memory locations; (d) the sparse tensor stored in the CISS format. Here a single CISS entry contains data from both PEs; thus memory accesses for both PEs in each cycle are done at contiguous memory locations. "**x**" denotes don't cares; (e) achieved bandwidth comparison between extended CSR and CISS for different numbers of PEs using a single channel DDR4 memory. Here the utilized bandwidth for CSR saturates at 1.9 GB/s for 8 PEs, while CISS is able to achieve 70% of the peak bandwidth.

5.1 Implementation Details of Tensaurus

Fig. 5 shows the architecture of *Tensaurus*, which consists of a 2-D array ($r \times c$) of compute PEs, a tensor load unit (TLU), a matrix load unit (MLU), an array of scratchpad memories (SPM) and a matrix store unit (MSU). TLU reads the first operand tensor, which is either stored in CISS format for sparse-dense kernels or dense format for dense-dense kernels from the main memory. MLU reads the dense operand matrices from the main memory and sends them to the SPMs. The SPMs receive the matrix data from the MLU and cache them in the double buffers. Each PE gets the data from the TLU and the SPM, performs VVMUL and VVADD operations in SIMD fashion (with vector length as VLEN) and accumulates the results in either a temporary shift register (TSR) or output shift register (OSR) depending on the type of accumulation. When the partial results for the current input tiles are completely evaluated, the PE drains the result to the MSU. Although each PE accumulates all the partial sums in local shift registers for the current input tiles, different input tiles may still update the same output element. Hence the MSU accumulates the drained results from the compute PEs and stores it in an output double buffer to perform reductions across different tiles. When all the input tiles for an output tile are processed, the MSU writes the results from the output buffer to main memory.

5.2 Implementation of SF^3 Compute Pattern

5.2.1 Tensor Load Unit (TLU)

The TLU reads the data for the first kernel operand from the main memory, which is either a sparse tensor stored in CISS format or a dense tensor stored in dense format. The read data is then pushed to the hardware queues connecting the TLU and the boundary PEs. The queues between the TLU and the boundary PEs ensure that the TLU and PEs can work asynchronously. To enable non-blocking memory accesses, the TLU is capable of handling out-of-order memory responses. It tries to send a load request to the main memory every clock cycle and pushes the request ID to a hardware queue in-order. When the response for a memory request arrives (out-of-order), the response data is written to the hardware queue and the corresponding entry is marked as completed. In each cycle, the TLU polls the head of the hardware queue and pops the data if marked as completed and sends it to the boundary PEs. Since the CISS format ensures that the data accessed by the PEs

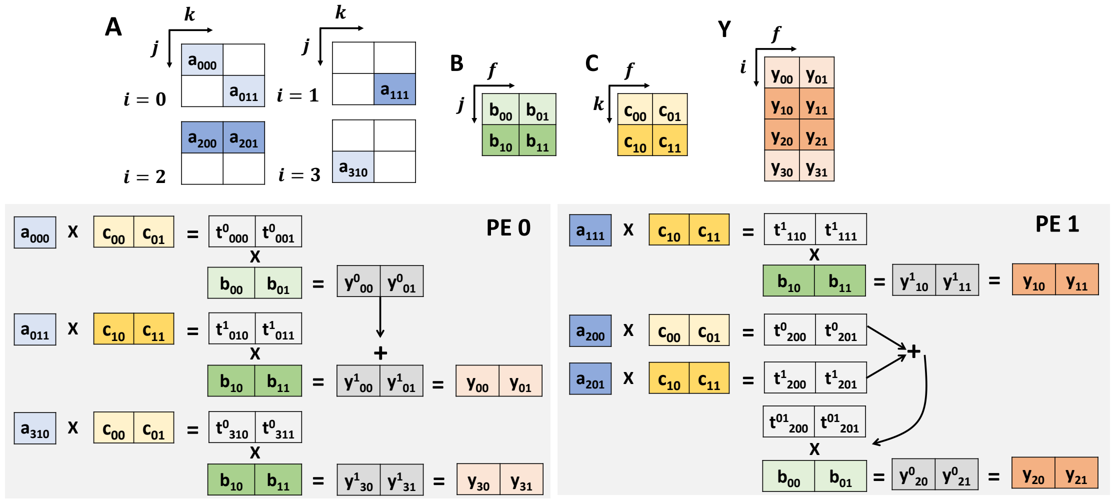


Figure 4: **Execution of SpMTTKRP on two PEs**.

at the same time is stored contiguously in the memory, the TLU sends wide read requests (one CISS entry) to the main memory to saturate the memory bandwidth.

5.2.2 *Matrix Load Unit (MLU)*

The MLU reads the data for the dense operand matrices from the main memory. Similar to the TLU, it consists of hardware queues to perform out-of-order memory reads. The MLU reads data in chunks of $c \cdot VLEN \cdot dw$ bits from the main memory and sends the data to the SPMs.

5.2.3 *Scratchpad Memories*

The SPMs are responsible for two major tasks: they read the matrix data from MLU and store it in the local buffers, and they serve a read request from the PEs in the corresponding column of the PE array. To avoid serialization between read and write to the buffers, the local buffers are implemented as double buffers. Each SPM receives data in chunks of $VLEN \cdot dw$ bits from the MLU and stores it into one of the buffers. The read and write port width of each buffer is $VLEN \cdot dw$ bits. Inside an SPM, a tile of a matrix is banked such that consecutive rows from the matrix are assigned to different buffers. Since a PE can request the data from any row of the matrix, a crossbar is used to connect different buffers to the PEs as shown in Fig. 5.

For SpMTTKRP and DMTTKRP, each SPM stores a tile of both the dense operand matrices **B** and **C**. For SpTTMc and DTTMc, only the SPM in the first column stores a tile of both the first and second dense operand matrices while the rest of the SPMs store only the tiles of the second operand matrix **C**. Thus, the SPM in the first column has 2× the amount of buffer capacity as compared to other SPMs. For SpMM and GEMM, each SPM stores tiles of the dense operand matrix **B**; and for SpMV and GEMV, only the SPM in the first column of PE array is active, and stores a tile of dense input vector **b**.

5.2.4 *Compute PEs*

The compute PEs are designed for efficient computation of the SF^3 compute pattern in Eq. (9). Fig. 5b shows the design of a single compute PE. It consists of a control processor (CP); two shift-registers: temporary shift register (TSR) and output shift register (OSR); a VVMUL unit; and a VVADD unit. The VVMUL and VVADD units process vectors of size VLEN. The TSR and OSR consist of TLEN and OLEN number of shift-registers, each of which is $VLEN \cdot dw$ bit wide. TSR is used to store the result of $\sum_{D_0} \mathbf{scalar} \cdot \mathbf{fiber}_0$ in Eq. (9) and OSR stores the partial sums for the output $\mathbf{fibers}_{out}$. Since for SpMM and SpMV, $\mathbf{fiber}_1$ and *op* are not applicable, TSR is not used and OSR stores the value of the computation $\sum_{D_0} \mathbf{scalar} \cdot \mathbf{fiber}_0$, which also is the $\mathbf{fibers}_{out}$. The PEs in the same row form a systolic array where the PEs on the left boundary read the (*scalar*,*j*,*k*) triplets (in the case of SpMTTKRP, DMTTKRP, SpTTMc and DTTMc) and (*scalar*,*j*) pairs (in the case of SpMM, GEMM, SpMV and GEMV), and forward it to the PEs in the same row. Here the *scalar* is the non-zero element from the sparse tensor operand.

For SpMTTKRP, each PE requests the data from the k^{th} row of the **C** matrix from the SPM. The SPM receives the request and sends VLEN elements from that row to the PE. The PE then replicates the *scalar* to perform a VVMUL operation corresponding to ① in Fig. 2a and accumulates the results in TSR. When all the non-zero entries in $\boldsymbol{\mathcal{A}}(i,j,:)$ are processed, the PE requests the SPM for the data from the j^{th} row of matrix **B**, performs a VVMUL operation with the partial results in TSR followed by a VVADD with the partial results in OSR ②, and accumulates the result in OSR. When all the non-zeros in the i^{th} slice of the tensor ($\boldsymbol{\mathcal{A}}(i,:,:)$) are processed, the results in the OSR are sent to the MSU, which writes them to the output buffer. Fig. 6 shows the execution of SpMTTKRP kernel for the sparse tensor in Fig. 3a on a 2×2 PE array. Here, the dense matrices are tiled along the columns, banked along rows and stored in different local buffers (VLEN is assumed to be one).

For SpTTMc, each PE acts in the exact same way as in SpMTTKRP; however, when all the non-zero entries in $\mathcal{A}(i,j,:)$ are processed and it has requested the data from the j^{th} row of matrix **B**, instead of directly performing VVMUL with TSR as in case of SpMTTKRP, it streams the values from the j^{th} row of matrix **B** one by one. It then replicates these values to perform VVMUL with TSR and

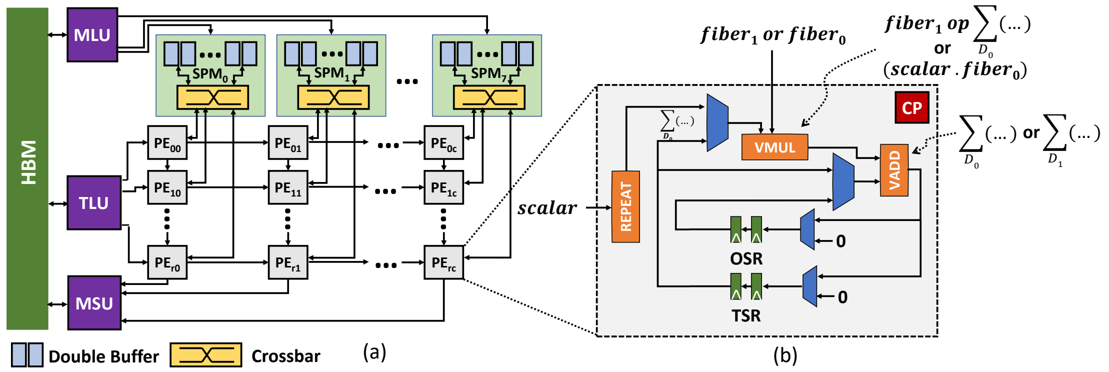


Figure 5: **Tensaurus architecture** – (a) architecture diagram of Tensaurus; (b) design of a single PE.

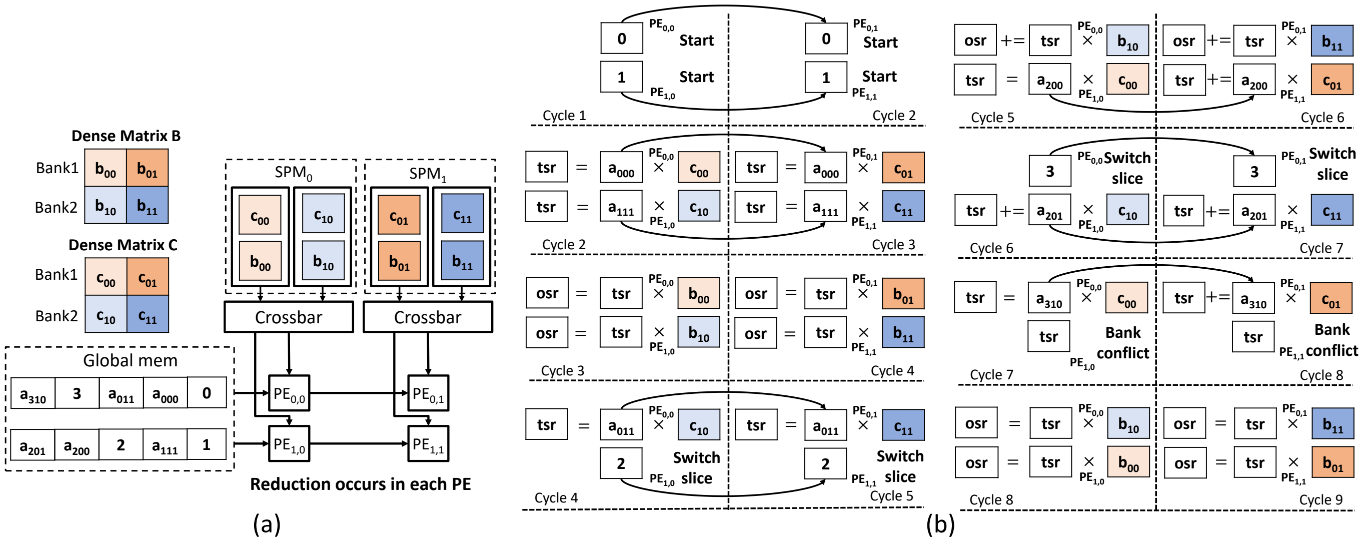


Figure 6: **SpMTTKRP on Tensaurus** – (a) 2×2 PE array in Tensaurus where dense matrix B is tiled, banked and stored in the buffers of different SPMs. The data from sparse tensor $\boldsymbol{\mathcal{A}}$ is stored in the memory in CISS format (only data values shown); (b) cycle by cycle execution of SpMTTKRP kernel on the 2×2 PE array. This is the exact same computation as the one shown in Fig. 4, however, here we chunk the vector computations (with VLEN=2) in PE0 of Fig. 4 into small chunks (with VLEN=1) and perform them in PE_{00} and PE_{01}. Similarly, the chunks of vector computations in PE1 are performed in PE_{10} and PE_{11}.

accumulates the results in one of the shift-registers in OSR. The number of shift registers in OSR ($OLEN$) is thus set to be VLEN. This approach in effect computes the outer-product of the row from matrix **B** and the value in TSR ②.

For SpMM, each PE sends the column index j to its SPM, which reads VLEN elements from the j^{th} row of matrix B and sends them to the PE. The PE then replicates the *scalar* value from sparse matrix, performs VVMUL operation corresponding to ① in Fig. 2c and accumulates the result in OSR. When all the non-zero entries in the current row of the sparse matrix $\mathbf{A}(i,:)$ are processed, the PE drains the OSR data to the MSU.

For SpMV, since the second operand is a dense vector instead of a dense matrix, only the first column of PEs in the systolic array is active. Each PE sends the column index j of the non-zero entry $\mathbf{A}(i,j)$ to the SPM similar to SpMM. However, this time the SPM reads only one element from the j^{th} index of the dense vector $\mathbf{b}$ and sends it to the PE. The PE then performs a scalar multiplication between the *scalar* value from sparse matrix $\mathbf{A}$ and the dense vector element, corresponding to ① in Fig. 2d, and accumulates the result in a single register of OSR. When all the non-zero entries in the current row of the sparse matrix are processed, the PE drains the OSR data to MSU.

For dense operations (DMTTKRP, DTTMc, GEMM and GEMV), the TLU reads the data in dense format, constructs a CISS representation on the fly and sends it to the PEs. The PEs and other units remain unaware that they are performing a dense computation. Since in the case of dense operations each PE from the same column would request the same entry from the SPM, these requests can get serialized by the crossbar. To avoid such serialization, for dense operations, only the PE in the first row is responsible for sending row addresses to the SPM and the crossbar broadcasts the response to all the PEs in the same column.

5.2.5 *Matrix Store Unit (MSU)*

The MSU is responsible for receiving the partial results from the PEs and accumulating them in the output buffer. The MSU drains the output buffer to the main memory

Table 2: Area and power breakdown of Tensaurus.

Component	**Area**(mm^2)	**%**	**Power (mW)**	**%**
PE	0.625	27.2 %	402.30	40.9 %
Xbar	0.066	2.8 %	24.27	2.5%
SPM	0.832	36.2 %	296.05	30.1 %
MSU	0.759	33.0 %	247.03	25.2 %
TLU	0.009	0.4 %	6.28	0.6%
MLU	0.009	0.4 %	6.28	0.6 %
Total	2.3	100 %	982.21	100 %

when all the input tiles corresponding to an output tile have been processed. Although buffering the intermediate results in the output buffer reduces the number of off-chip memory accesses, it also limits the tile size of the sparse input tensor. Since for very sparse tensors the benefit of storing the intermediate results in the buffer is outweighed by the larger tile size of the tensor (as it results in more reuse of the dense operand matrices), the MSU can be configured to directly accumulate the results in the main memory.

6. EXPERIMENTAL SETUP

Simulation Infrastructure – To evaluate the performance of Tensaurus, we model our architecture consisting of TLU, MLU, SPMs, PEs, MSU and HBM using the gem5 simulator [38]. We use an 8×8 PE array with $VLEN = 4$. Each SPM except the first consists of a double buffer of size $2 \times 16KB$ where each side of the double buffer is divided into 8 $2KB$ banks. The SPM in the first column has a double buffer of size $2 \times 32KB$ divided into 16 $2KB$ banks. The MSU consists of an output double buffer of size $2 \times 128KB$, which is further divided into 8 $16KB$ banks. For HBM, we use the gem5 model, which supports up to 8 128-bit physical channels, runs at 1GHz clock frequency and provides a peak memory bandwidth of 128 GB/s. Tensaurus is attached to a CPU as a co-processor, where the CPU executes instructions to configure Tensaurus to run a specific tensor kernel. The configuration instructions configure Tensaurus for: (1) mode of operation like SpMTTKRP, SpMM, etc. and (2) size of tensors and matrices.

Measurements – We implemented the PEs and the crossbar in RTL using PyMTL [39] and synthesized them using the Synopsys Design Compiler using TSMC 28nm library and placed-and-routed using Cadence Innovus. For the SPM and MSU, since the majority of area and power is dominated by scratchpads, we used CACTI 7.0 [40] to model SRAM latencies, area and power. For TLU, we pessimistically assumed the same area and power as a single PE. Table 2 shows the area and power breakdown of different components of the design. For HBM we use the energy numbers from Shilov *et al.* [41].

Baselines – We compare our design against four baselines: CPU, GPU, Cambricon-X [35] and T2S-Tensor [17].

CPU: We use SPLATT [20] and Sparse BLAS [42] to evaluate our benchmarks on a single core of an Intel(R) Xeon(R) CPU E7-8867 running at 2.40 GHz with 32 KB L1 cache, 256 KB L2 cache and 45 MB of L3 cache. For energy estimates we use McPAT 1.3 [43] CPU energy models.

Table 3: Tensors with their dimensions, number of non-zeros (nnz), density and problem domain.

Tensor	**Dimensions**	**nnz**	**Density**	**Domain**
nell-2	12K × 9K × 28K	77M	2.5e-5	NLP
netflix	480K × 18K × 2K	100M	5.7e-6	Rec. Sys.
poisson3D	3K × 3K × 3K	99M	3.6e-3	Synthetic

Table 4: Weight matrices from AlexNet and VGG-16 with their dimensions, number of non-zeros (nnz) and density.

	Layer	**Dim.**	**Density**	**Layer**	**Dim.**	**Density**
AlexNet	c1	96 × 363	0.84	c2	256 × 1200	0.38
	c3	384 × 2304	0.35	c4	384 × 1728	0.37
	c5	256 × 1728	0.37	fc6	9216 × 4096	0.09
	fc7	4096 × 4096	0.09	fc8	4096 × 1000	0.25
VGG-16	c1_1	64 × 27	0.58	c1_2	64 × 576	0.22
	c2_1	128 × 1152	0.34	c2_2	128 × 1152	0.36
	c3_1	256 × 1152	0.53	c3_2	256 × 2304	0.24
	c3_3	256 × 2304	0.42	c4_1	512 × 2304	0.32
	c4_2	512 × 4608	0.27	c4_3	512 × 4608	0.34
	c5_1	512 × 4608	0.35	c5_2	512 × 4608	0.29
	c5_3	512 × 4608	0.36	fc6	25088 × 2096	0.01
	fc7	4096 × 4096	0.02	fc8	4096 × 1000	0.09

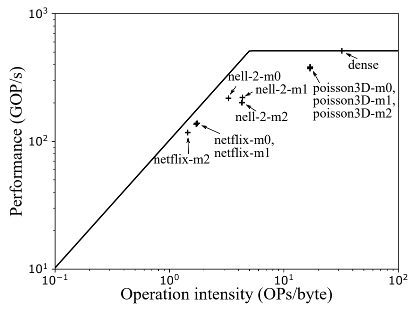
GPU: We use ParTI [44,45] and cuSPARSE [46] to evaluate the benchmarks on a modern GPU Titan Xp, which has GDDR5x DRAM with a peak bandwidth of 547.6 GB/s and a peak 32-bit performance of 12.15 TFLOP/s. We use CUDA 9.1 for programming the GPU. For power estimation, we use thermal design power (TDP) from the GPU datasheet.

Accelerator: For SpMM, we also compare our work against the Cambricon-X [35] state-of-the-art CNN accelerator, which uses sparse weights and dense activations. We implement the architecture of Cambricon-X in gem5 and scale it to have the same bitwidth, clock frequency, number of multiply-accumulate (MAC) units, size of on-chip RAM and DRAM bandwidth as our accelerator. For energy comparisons, we use the power numbers from [35] which are in 65nm technology node and scale them to 28nm using [47,48]. For DRAM energy, we measure the number of DRAM accesses from our simulator and use the HBM energy from Shilov *et al.* [41].

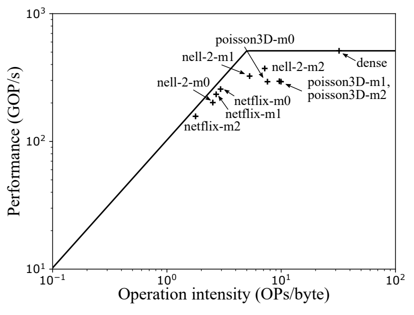
For DMTTKRP, DTTMc and GEMM we compare our design against T2S-Tensor [17], which implements these kernels on an FPGA. We scale their design to use the same number of MAC units and clock frequency as our design.

Datasets – For SpMTTKRP and SpTTMc, we use the tensor datasets shown in Table 3. NELL-2 tensor is a snapshot of the Never Ending Language Learner knowledge base that attempts to create a computer system that learns how to read the web [16]. The Netflix dataset is taken from Netflix Prize competition [15] and Poisson3D is taken from [27]. NELL-2 and Netflix are public datasets and taken from Smith *et al.* [50].

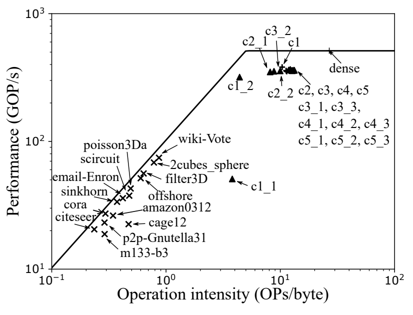
For SpMM, we use the pruned models for AlexNet and VGG-16 [51]. We did not use any of the newer CNN models for this study as their pruned weights are not publicly available. Table 4 shows the sparse weight matrices in these CNN models with their size and densities. For SpMM, we also use the sparse matrices from SuiteSparse [49] and graph benchmarks from GraphSAGE [28]. Table 5 shows



(a) Roofline for SpMTTKRP.



(b) Roofline for SpTTMc.



(c) Roofline for SpMM.

Figure 7: **Roofline evaluation of SpMTTKRP, SpTTMc and SpMM on Tensaurus** – the x-axis shows operation intensity which is number of operations (multiply and add) performed for each byte of data accessed from the off-chip memory; the y-axis shows the performance in GOP/s.

Table 5: Matrices from SuiteSparse [49] with their dimensions, number of non-zeros (nnz), density and problem domain.

Matrix	**Dim**	**nnz**	**Density**	**Domain**
amazon0312	401K × 401K	3.2M	1.9e-5	Copurchase network
m133-b3	200K × 200K	801K	2.0e-5	Combinatorics
scircuit	171K × 171K	959K	3.2e-5	Circuit simulation
p2pGnutella31	63K × 63K	148K	3.7e-5	p2p network
offshore	260K × 260K	4.2M	6.2e-5	EM Problem
cage12	130K × 130K	2.0M	1.1e-4	Weighted graph
2cubes-sphere	101K × 101K	1.6M	1.5e-4	EM Problem
filter3D	106K × 106K	2.7M	2.4e-4	Reduction problem
emailEnron	36.7K × 36.7K	368K	2.7e-4	Email network
citeseer	3.3K × 3.3K	4.7K	4.2e-4	Graph Learning
cora	2.7K × 2.7K	5.3K	7.2e-4	Graph Learning
wikiVote	8.3K × 8.3K	104K	1.5e-3	Wikipedia network
poisson3Da	14K × 14K	353K	1.8e-3	Fluid Dynamics

the sparse matrices from SuiteSparse and GraphSAGE. For SpMV, we use the same matrices from SuiteSparse and GraphSAGE as in SpMM.

7. EVALUATION

7.1 Roofline Evaluation

Fig. 7 shows the throughput of SpMTTKRP, DMTTKRP, SpTTMc, DTTMc, SpMM, and GEMM, under the roofline [52] of our accelerator. The horizontal line towards the right of the plot shows the peak attainable performance from the design when the operation intensity is high (kernel is compute bound) and the inclined line (with slope 1) towards the left shows the peak attainable performance when the operation intensity is low (kernel is memory bound). The value of throughput for operation intensity of 1 represents the peak memory bandwidth (in GB/s). The gap between the roofline and the achieved performance of a kernel indicates the inefficiencies within the hardware. Our design consists of an 8×8 PE array, each with 4 SIMD MAC units and hence it has $8\times8\times4\times2=512$ scalar multipliers and adders. Since we simulate our design for a 2GHz clock frequency, assume that the scratchpads are synchronous, and that each PE spends every other clock cycle to access the scratchpads instead of doing a MAC, the peak attainable throughput is $512\times2\times0.5=512$ GOP/s. For peak memory bandwidth, we use the peak bandwidth of HBM1 which is 128 GB/s.

Fig. 7a shows the achieved throughput for SpMTTKRP along all the three modes of the three tensors shown in Table 3. Here, SpMTTKRP is memory bound for all the tensors except `poisson3D`, where it is compute bound due to the highest density of `poisson3D` among all the tensors. For all the SpMTTKRP kernels Tensaurus is able to perform close to the peak throughput.

Fig. 7b shows the achieved throughput for SpTTMc for the three tensors along each mode. Here, `nell-2-m0`, `nell-2-m1` and `poisson3D` are compute bound while the others are memory bound and the achieved performance on each kernel is very close to the peak throughput. It can also be seen from Figs. 7a and 7b that the operation intensity for the same tensor along the same mode is higher for SpTTMc as compared to SpMTTKRP. The reason behind this is SpTTMc performs a Kronecker product as an intermediate operation as shown in Fig. 2d, which has more MAC operations compared to the Hadamard product in Fig. 2c.

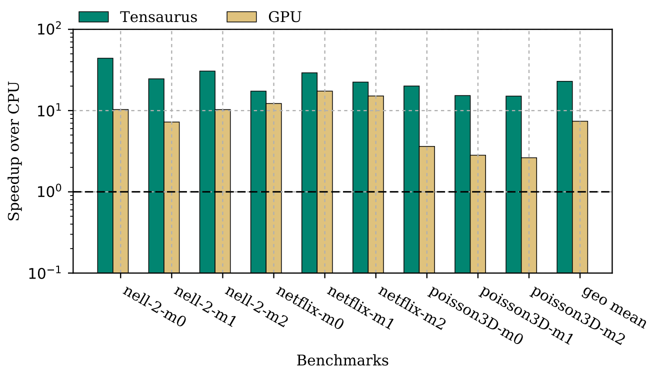
Fig. 7c shows the achieved throughput for sparse convolution layers in AlexNet and VGG-16 and sparse matrices from SuiteSparse [49] and GraphSAGE [28] for the SpMM kernel. For all the layers except c1_1 and c1_2 the achieved throughput is very close to the peak throughput. For c1_1 and c1_2, since the sparse weight matrices are very small (Table 4), the scratchpads and MAC units in Tensaurus are underutilized. For the SuiteSparse and GraphSAGE matrices since the densities of these matrices are very low (Table 5), the SpMM kernel is memory bound and Tensaurus achieves very close to the peak throughput in the memory bound region.

Figs. 7a, 7b and 7c also show the achieved throughput for DMTTKRP, DTTMc and GEMM on our accelerator (labeled as "dense"). It can be seen that all the dense kernels are compute bound and our accelerator achieves close to the peak throughput for each of them.

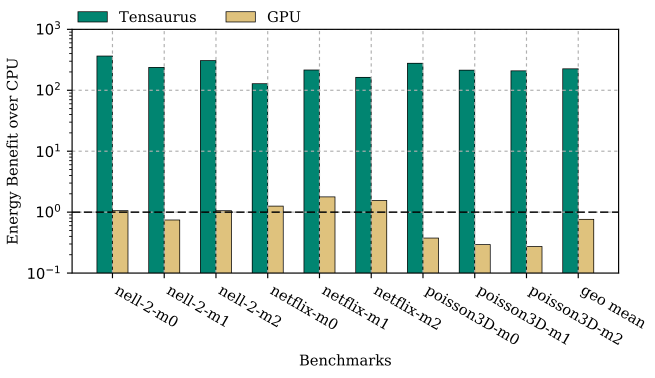
7.2 Performance Evaluation

Fig. 8a shows the speedup of Tensaurus and GPU (ParTI) on SpMTTKRP for the three tensors along each mode over the CPU (SPLATT) baseline. Tensaurus achieves a geomean speedup of 22.9× over CPU and 3.1× over GPU.

Fig. 9a shows the speedup of Tensaurus, and GPU (ParTI) for SpTTMc on three tensors along each mode over the CPU (SPLATT) baseline. Here, Tensaurus achieves

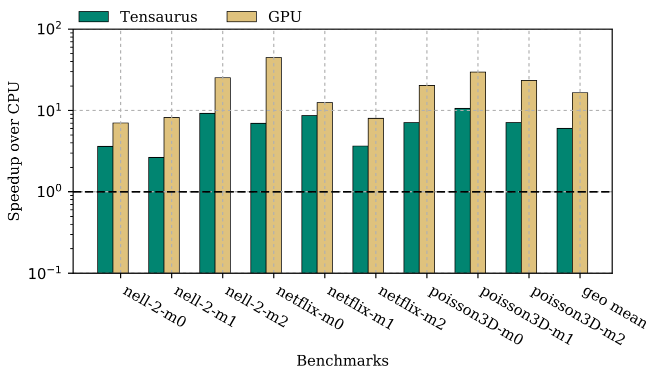


(a) **Speedup for SpMTTKRP.**

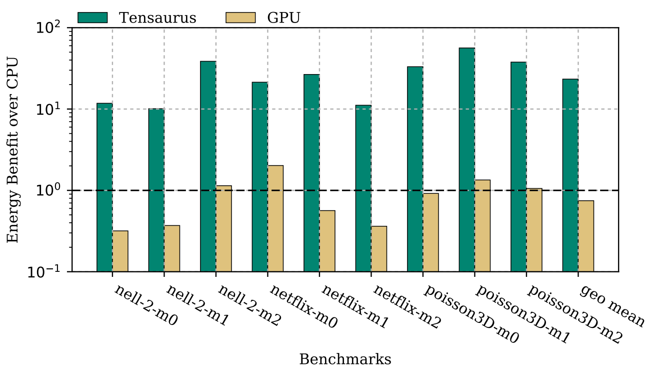


(b) **Energy benefit for SpMTTKRP.**

Figure 8: **Speedup and energy benefit of Tensaurus and GPU (PaRTI) over CPU (SPLATT) baseline for SpMTTKRP on sparse tensors in Table 3.**



(a) **Speedup for SpTTMc.**



(b) **Energy benefit for SpTTMc.**

Figure 9: **Speedup and energy benefit of Tensaurus and GPU (PaRTI) over CPU (SPLATT) baseline for SpTTMc on sparse tensors in Table 3.**

6.02× speedup over CPU. However, Tensaurus achieves 0.1× of the performance of the GPU baseline (PaRTI); in PaRTI, a significant portion of the SpTTMc algorithm runs on the host CPU, but for comparison with Tensaurus we do not take into account the CPU execution time. After taking CPU execution time into account, Tensaurus would achieve a 5× speedup over GPU. Unlike SpMTTKRP, where the speedup of Tensaurus is more than 20× over CPU, we also achieve a lower speedup of 6.02× in the case of SpTTMc. The reason behind this is SpTTMc benefits significantly from the operand factorization as discussed in Section 2. A smaller tile size and on-chip memory limits operand factoring opportunities, and Tensaurus uses just 512KB of on-chip memory as compared to 45MB of L3 cache in the case of the CPU.

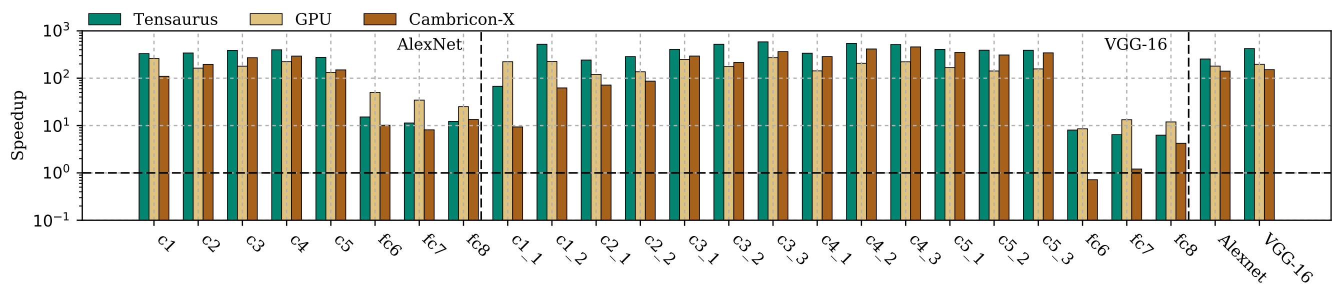
Fig. 10a shows the speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over the CPU (SparseBLAS) baseline for AlexNet and VGG-16. For most of the convolution layers Tensaurus performs better than all the baselines. On average, Tensaurus is 349.2×, 1.8× and 1.9× faster than CPU, GPU, and Cambricon-X, respectively.

Fig. 11a shows the speedup of Tensaurus, GPU and Cambricon-X over CPU baseline for benchmarks from SuiteSparse and GraphSAGE. Unlike CNNs where the matrices have low sparsity (high density), these matrices have really high sparsity (low density). Tensaurus performs better than Cambricon-X on all the matrices and often beats GPU. Overall Tensaurus achieves 125.8× and 119.7× speedup over CPU and Cambricon-X, respectively and achieves 0.87× of the performance of GPU for these matrices.

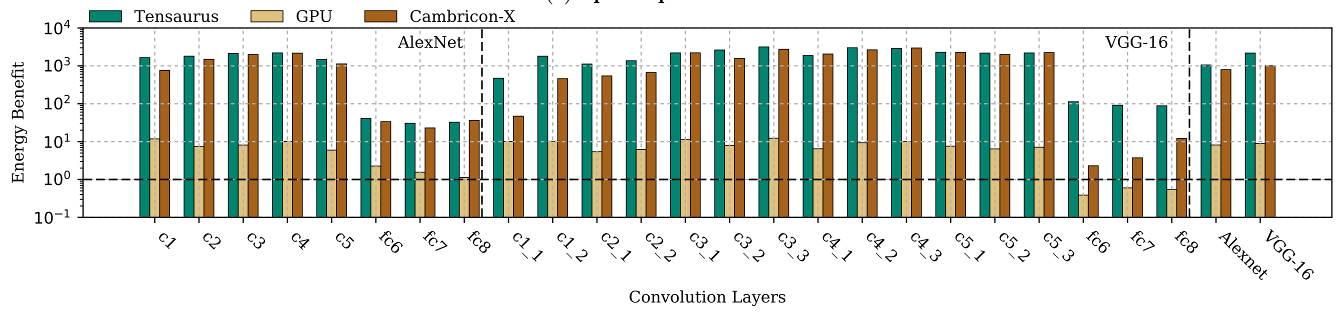
To further analyse the performance of Tensaurus for different densities of sparse matrix, we generate synthetic matrices and measure SpMM performance for Tensaurus and all the baselines. Fig. 13 shows the speedup of Tensaurus, GPU and Cambricon-X over CPU baseline for different densities of sparse matrix. As it can be seen, Tensaurus performs consistently better than all the baselines and the performance of GPU is very similar to the performance of Tensaurus.

Fig. 12a shows the speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (SparseBLAS) baseline for benchmarks from SuiteSparse and GraphSAGE for SpMV kernel. Overall Tensaurus achieves 7.7× and 0.45× speedup over CPU and GPU. Since SpMV is highly memory bound and GPU has 5× more bandwidth and more on-chip memory the performance of SpMV is better on GPU as compared to Tensaurus.

We further compare the performance of DMTTKRP, DTTMc and GEMM with T2S-Tensor. Table 6 shows the throughput of our accelerator in dense mode (Tensaurus-dense) compared to T2S-Tensor. As it can be seen for DMTTKRP, DTTMc and GEMM, Tensaurus-dense achieves close to 0.5× of the performance of T2S-Tensor, which is



(a) **Speedup over CPU.**



(b) **Energy benefit over CPU.**

Figure 10: **Speedup and energy benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) baseline for SpMM (convolution layers) and SpMV (fully connected layers) on sparse matrices from AlexNet and VGG-16.**

a pessimistic estimate since we assume perfect scaling for T2S-Tensor.

Table 6: Comparison between the performance of Tensaurus-dense and T2S-Tensor [17].

Benchmark	Throughput (GOP/s)		Speedup
	Tensaurus-dense	T2S-Tensor	
DMTTKRP	511.9	986.3	0.52×
DTTMc	498.9	926.6	0.54×
GEMM	506.5	1019.8	0.49×

7.3 Energy Evaluation

Figs. 8b and 9b show the energy benefit of our accelerator and GPU over the CPU baseline for SpMTTKRP and SpTTMc on three tensors along each mode. Overall our accelerator is 223.2× and 292.8× more energy efficient than CPU and GPU for SpMTTKRP and 23.2× and 30.9× more energy efficient than CPU and GPU for SpTTMc.

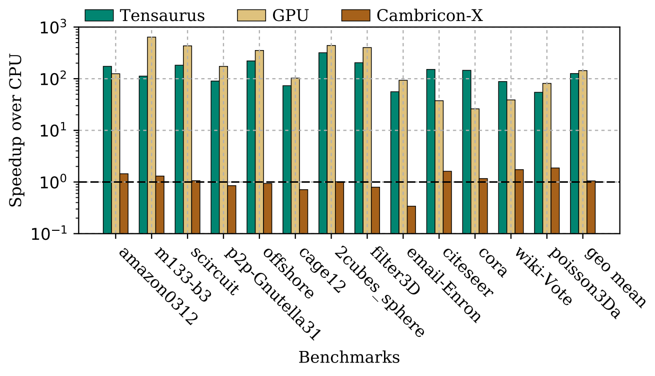
Fig. 10b shows the energy benefit of our accelerator, GPU and Cambricon-X over the CPU baseline for AlexNet and VGG-16. On average, our accelerator is 1983.7×, 226.6× and 1.7× more energy efficient than CPU, GPU and Cambricon-X. Fig. 11b shows the energy benefit of our accelerator for SpMM on SuiteSparse and GraphSAGE matrices. Overall our accelerator is 405.6×, 62.5×, and 101.5× more energy efficient than CPU, GPU and Cambricon-X.

Fig. 12b shows the energy benefit of our accelerator and GPU over the CPU baseline for SpMV on matrices from SuiteSparse. For SpMV, our accelerator is 46.4× and 60.1× more energy efficient than CPU and GPU.

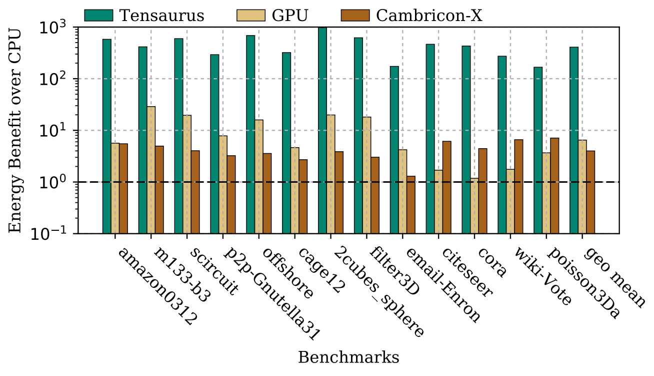
8. RELATED WORK

Sparse Storage Formats – Many sparse storage formats have been proposed in the literature. CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) and COO (Co-ordinate) are the most commonly used sparse storage formats for CPUs. Liu *et al.* [53] proposed a sparse tensor storage format F-COO, which is similar to the co-ordinate format and used it for GPUs. CSF [20] and Hi-COO [21] are other sparse tensor storage formats that are are based on CSR and COO, respectively. OuterSPACE [36] uses a variant of CSR and CSC formats called CR and CC for sparse-sparse matrix-matrix multiplication (SpGEMM). For machine learning hardware, researchers have proposed multiple variants of CSR and CSC formats. For example, Cambricon-X [35] proposed a modification of CSR format where the non-zeros are compressed and stored in contiguous memory and index vectors are used to decode the row and column indices. EIE [54] uses a variant of CSC storage format where instead of storing the row indices they store the number of zeros before a non-zero element. However, since these works focus on deep learning, especially CNNs, their sparse storage format is specialized for low sparsity (high density).

Software Frameworks – TACO [55] is a language and compiler framework to generate high-performance code for sparse matrix and tensor kernels for CPUs. Kjolstad *et al.* [56] introduced workspace optimizations in TACO to implement operand factoring optimizations in tensor kernels. SPLATT [20,27] introduced a C library implementing SpMTTKRP and SpTTMc with shared memory parallelization. Bhaskaran *et al.* [57] proposed various techniques to reduce memory usage and execution time for sparse tensor factorization algorithms. Ballard *et al.* [19] and Choi

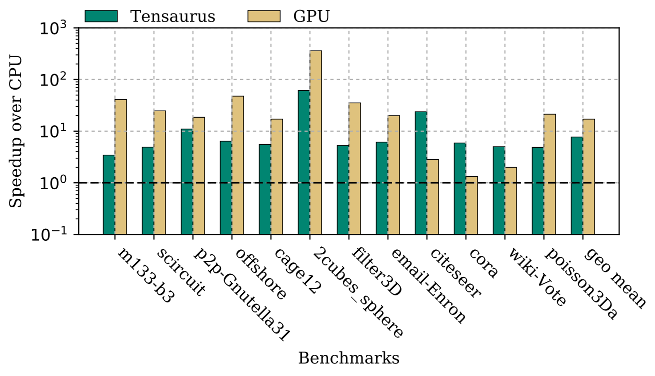


(a) **Speedup for SpMM.**

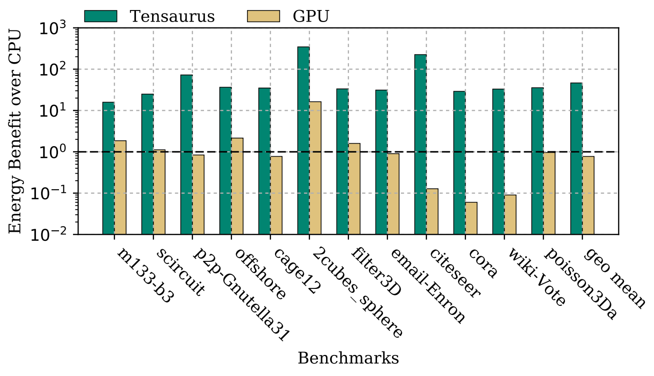


(b) **Energy benefit for SpMM.**

Figure 11: **Speedup and energy benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) baseline for SpMM on sparse matrices from SuiteSparse and GraphSAGE.**



(a) **Speedup for SpMV.**



(b) **Energy benefit for SpMV.**

Figure 12: **Speedup and energy benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) baseline for SpMV on sparse matrices from SuiteSparse and GraphSAGE.**

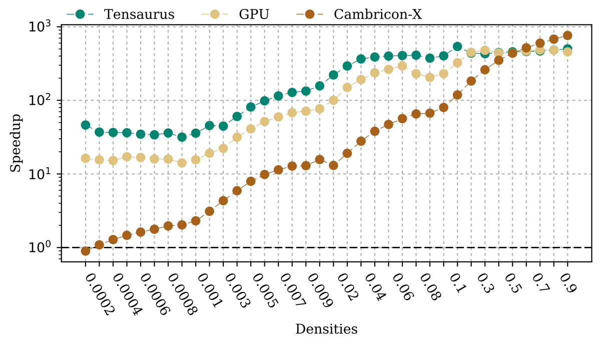


Figure 13: **Speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) baseline for SpMM on synthetic matrices with density varying from 0.0001 to 0.9.**

et al. [26] proposed methods to perform DMTTKRP and DTTMc on CPU and GPU, respectively.

Hardware Accelerators – Srivastava *et al.* [17] proposed a language and compilation framework called T2S-Tensor to generate high performance hardware for dense tensor computations such as GEMM, DMTTKRP and DTTMc. Zhang *et al.* [18] proposed a hardware accelerator for DTTMc. Hegde *et al.* [58] proposed a hardware accelerator called ExTensor for sparse tensor algebra using the ideas of merge lattice proposed in TACO [55]. This work, however, does not accelerate sparse tensor factorizations as they involve more than two operands and their performance are significantly affected by operand factoring optimizations. ExTensor also uses significantly more on-chip storage than Tensaurus (30 MB as compared to 0.5 MB). This allows ExTensor to read an entire tile of the sparse input tensor on-chip in a streaming manner, avoiding the need for a special sparse storage format. However, this comes at the cost of area and energy where ExTensor is 40× larger than Tensaurus for the same technology node. Kanellopoulos *et al.* proposed a hardware-software cooperative mechanism to accelerate sparse matrix operations. For SpMM and GEMM, prior works involving ASIC implementations include Cambricon-X [35], Cambricon-S [59], Cnvlutin [60], SCNN [61], SparTen [62] and OuterSPACE [36]. Cambricon-X [35] and Cambricon-S [59] implement hardware accelerators for SpMM and SpGEMM in CNNs where either the weight matrices or both weight matrices and neurons are sparse. SCNN [61] proposes a SpGEMM accelerator for CNNs which can exploit the sparsity in both weights and neurons. OuterSPACE [36] proposed an accelerator design for SpGEMM. EIE [54] proposes the SpMSpV (sparse matrix sparse vector multiplication) accelerator for fully connected layers in CNN and show significant performance gains over CPU and GPU. TPU [63] implemented a 2-d systolic array for GEMM. Prior work involving FPGA implementations for sparse-dense and sparse-sparse matrix-

matrix and matrix-vector accelerators include [64], ESE [65] and [37]. Lu *et al.* [64] proposed a CNN accelerator with sparse weights. ESE [65] proposed an FPGA-accelerator for SpMV in LSTMs. Fowers *et al.* [37] proposed SpMV accelerator for sparse matrices.

9. CONCLUSION

In this work, we propose a new sparse storage format which allows accessing sparse data in vectorized manner and co-design a hardware accelerator for sparse and dense tensor factorizations. We extracted a common compute pattern among different tensor factorizations and matrix operations, and implemented the pattern in hardware. With such hardware software co-design we achieve significant speedup and energy benefit over multiple hardware and software baselines.

Acknowledgement

We thank Ayoub Benkhoris and Congyang Li for their contributions to the RTL simulation and verification. We also appreciate the help from Ritchie Zhao, who provided the models for CNN benchmarks. This research was funded in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, under NSF Awards #1453378, #1512937, #1909661, NSF/Intel CAPA Award #1723773, and by AFRL and DARPA under agreement number FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

10. REFERENCES

[1] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, “Tensor Decomposition for Signal Processing and Machine Learning,” *IEEE Trans. on Signal Processing*, 2017.

[2] 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*, 2015.

[3] M. A. O. Vasilescu and D. Terzopoulos, “Multilinear Analysis of Image Ensembles: Tensorfaces,” *European Conf. on Computer Vision*, 2002.

[4] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, “ParCube: Sparse Parallelizable Tensor Decompositions,” *Joint European Conf. on Machine Learning and Knowledge Discovery in Databases*, 2012.

[5] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen, “CubeSVD: A Novel Approach to Personalized Web Search,” *Int’l Conf. on World Wide Web*, 2005.

[6] T. Kolda and B. Bader, “The TOPHITS Model for Higher-order Web Link Analysis,” *Workshop on Link Analysis, Counterterrorism and Security*, 2006.

[7] J. C. Ho, J. Ghosh, and J. Sun, “Marble: High-throughput Phenotyping from Electronic Health Records via Sparse Nonnegative Tensor Factorization,” *Int’l Conf. on Knowledge Discovery and Data Mining*, 2014.

[8] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges,” *IEEE Signal Processing Magazine*, 2018.

[9] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation,” *Int’l Conf. on Neural Information Processing Systems (NIPS)*, 2014.

[10] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky, “Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition,” *arXiv preprint arXiv:1412.6553*, 2014.

[11] T. G. Kolda and B. W. Bader, “Tensor Decompositions and Applications,” *SIAM Review*, 2009.

[12] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep Learning with COTS HPC Systems,” *Int’l Conf. on Machine Learning*, 2013.

[13] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, “Neural Network Stream Processing Core (NnSP) for Embedded Systems,” *Int’l Symp. on Circuits and Systems*, 2006.

[14] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the Speed of Neural Networks on CPUs,” *Workshop on Deep Learning and Unsupervised Feature Learning, NIPS*, 2011.

[15] J. Bennett, S. Lanning, *et al.*, “The Netflix Prize,” *Proceedings of KDD Cup and Workshop*, 2007.

[16] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, “Toward an Architecture for Never-Ending Language Learning.,” *AAAI*, 2010.

[17] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. H. Herr, C. Hughes, T. Mattson, and P. Dubey, “T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations,” *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[18] K. Zhang, X. Zhang, and Z. Zhang, “Tucker Tensor Decomposition on FPGA,” *arXiv preprint arXiv:1907.01522*, 2019.

[19] G. Ballard, K. Hayashi, and K. Ramakrishnan, “Parallel Nonnegative CP Decomposition of Dense Tensors,” *Int’l Conf. on High Performance Computing (HiPC)*, 2018.

[20] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, “SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication,” *Int’l Parallel and Distributed Processing Symp. (IPDPS)*, 2015.

[21] J. Li, J. Sun, and R. Vuduc, “HiCOO: Hierarchical Storage of Sparse Tensors,” *Int’l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2018.

[22] J. Choi, X. Liu, S. Smith, and T. Simon, “Blocking Optimization Techniques for Sparse Tensor Computation,” *Int’l Parallel and Distributed Processing Symp. (IPDPS)*, 2018.

[23] L. De Lathauwer, B. De Moor, and J. Vandewalle, “On The Best Rank-1 and Rank-(r 1, r 2,..., rn) Approximation of Higher-Order Tensors,” *SIAM Journal on Matrix Analysis and Applications*, 2000.

[24] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, “On Optimizing Distributed Tucker Decomposition for Dense Tensors,” *Int’l Parallel and Distributed Processing Symp. (IPDPS)*, 2017.

[25] G. Ballard, A. Klinvex, and T. G. Kolda, “TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Tensor Decomposition,” *arXiv preprint arXiv:1901.06043*, 2019.

[26] J. Choi, X. Liu, and V. Chakaravarthy, “High-performance dense tucker decomposition on GPU clusters,” *Int’l Conf. for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[27] S. Smith and G. Karypis, “Accelerating the Tucker Decomposition with Compressed Sparse Tensors,” *European Conference on Parallel Processing*, 2017.

[28] W. Hamilton, Z. Ying, and J. Leskovec, “Inductive Representation Learning on Large Graphs,” *Advances in Neural Information Processing Systems*, 2017.

[29] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” *arXiv preprint arXiv:1609.02907*, 2016.

[30] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient Primitives for Deep Learning,” *arXiv preprint arXiv:1410.0759*, 2014.

[31] S. Brin and L. Page, “The Anatomy of a Large-Scale Hypertextual Web Search Engine,” *Computer Networks and ISDN Systems*, 1998.

[32] E. Nurvitadhi, A. Mishra, and D. Marr, “A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine,” *Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2015.

[33] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr, “Fine-Grained Accelerators for Sparse Machine Learning Workloads,” *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2017.

[34] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks,” *Int'l Symp. on Parallelism in Algorithms and Architectures*, 2009.

[35] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-x: An Accelerator for Sparse Neural Networks,” *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[36] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator,” *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2018.

[37] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, “A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication,” *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[38] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, “The gem5 Simulator,” *ACM SIGARCH Computer Architecture News*, 2011.

[39] D. Lockhart, G. Zibrat, and C. Batten, “PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research,” *Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[40] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “CACTI 6.0: A Tool to Model Large Caches,” *HP laboratories*, 2009.

[41] A. Shilov., “JEDEC Publishes HBM2 Specification.” `http://www.anandtech.com/show/9969/jedec-publisheshbm2-specification`, 2016.

[42] I. S. Duff, M. A. Heroux, and R. Pozo, “An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum,” *ACM Trans. on Mathematical Software (TOMS)*, 2002.

[43] 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,” *Int'l Symp. on Microarchitecture (MICRO)*, 2009.

[44] J. Li, Y. Ma, and R. Vuduc, “ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs,” 2018.

[45] 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*, 2019.

[46] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “CuSPARSE Library,” *GPU Technology Conference*, 2010.

[47] “28 nm lithography process.” `https://en.wikichip.org/wiki/28_nm_lithography_process`.

[48] “65 nm lithography process.” `https://en.wikichip.org/wiki/65_nm_lithography_process`.

[49] T. A. Davis and Y. Hu, “The University of Florida Sparse Matrix Collection,” *ACM Trans. on Mathematical Software (TOMS)*, 2011.

[50] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis, “The Formidable Repository of Open Sparse Tensors and Tools.” `http://frostt.io/tensors/`, 2017.

[51] S. Han, J. Pool, J. Tran, and W. Dally, “Learning Both Weights and Connections for Efficient Neural Network,” *Advances in Neural Information Processing Systems*, 2015.

[52] S. Williams, A. Waterman, and D. Patterson, “Roofline: An Insightful Visual Performance Model for Multicore Architectures,” *Commun. ACM*, 2009.

[53] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, “A Unified Optimization Approach for Sparse Tensor Operations on GPUs,” *Int'l Conf. on Cluster Computing (CLUSTER)*, 2017.

[54] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[55] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The Tensor Algebra Compiler,” *Intl'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

[56] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, “Sparse Tensor Algebra Optimizations with Workspaces,” *arXiv preprint arXiv:1802.10574*, 2018.

[57] M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin, “Memory-Efficient Parallel Tensor Decompositions,” *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[58] 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,” *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[59] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, “Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach,” *Int'l Symp. on Microarchitecture (MICRO)*, 2018.

[60] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing,” *ACM SIGARCH Computer Architecture News*, 2016.

[61] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks,” *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[62] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, “SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks,” *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[63] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[64] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, “An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs,” *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[65] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA,” *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.