Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory

Abstract

Sparse tensor contractions appear commonly in many applications. Efficiently computing a two sparse tensor product is challenging: It not only inherits the challenges from common sparse matrix-matrix multiplication (SpGEMM), i.e., indirect memory access and unknown output size before computation, but also raises new challenges because of high dimensionality of tensors, expensive multi-dimensional index search, and massive intermediate and output data. To address the above challenges, we introduce three optimization techniques by using multi-dimensional, efficient hashtable representation for the accumulator and larger input tensor, and all-stage parallelization. Evaluating with 15 datasets, we show that Sparta brings 28 − 576× speedup over the traditional sparse tensor contraction with SPA. With our proposed algorithm- and memory heterogeneity-aware data management, Sparta brings extra performance improvement on the heterogeneous memory with DRAM and Intel Optane DC Persistent Memory Module (PMM) over a state-of-the-art software-based data management solution, a hardware-based data management solution, and PMM-only by 30.7% (up to 98.5%), 10.7% (up to 28.3%) and 17% (up to 65.1%) respectively.

1 Introduction

Tensors, especially those high-dimensional sparse tensors are attracting increasing attentions, because of their popularity in many applications. High-order sparse tensors have been studied well in tensor decomposition on various hardware platforms [12, 29, 37–40, 43, 52, 53, 56, 63–65] with a focus on the product of a sparse tensor and a dense matrix or vector.

Nevertheless, the two sparse tensor contraction (SpTC), foundation for a spectrum of applications, such as quantum chemistry, quantum physics and deep learning [7, 20, 33, 41, 58, 59], are still lack of sufficient research, especially with element-/pair-wise sparsity. In essence, SpTC, a high-order extension of sparse matrix-matrix multiplication (SpGEMM), multiplies two sparse tensors along with their common dimensions. Efficient SpTC introduces multiple challenges.

First, the size and non-zero pattern of the output tensor are unknown before computation. Thus, memory allocation for the output tensor is difficult. Unlike operations such as a sparse tensor multiplies a dense matrix/vector where the size of the output data is predictable, the output tensor of an SpTC is usually sparse and the non-zero pattern (e.g., the number of non-zero elements and their distribution) is unpredictable before the actual computation. Sparse data objects and unpredictable output size also exist in SpGEMM. Two popular approaches have been proposed to solve these issues for SpGEMM while are not efficient for SpTC. The first approach, using an extra symbolic phase [49] to predict the accurate output size and non-zero pattern, suffers from expensive pre-processing and is unaffordable in a dynamic sparsity environment. This issue is especially severe in SpTC, because an SpTC with the exactly same input is usually computed only once in a long sequence of tensor contractions [7]. However, with the symbolic approach, every SpTC is attached to both a symbolic phase and SpTC computation, which is very expensive, especially for large applications. The second approach makes a loose upper-bound prediction on the memory consumption of the output tensor. However, a tight prediction for SpTC of high-order tensors is very difficult because its more contract dimensions (see Section 2.2) make the prediction less accurate based on the existed prediction algorithms [5, 14].

Second, irregular memory accesses along with multi-dimensional index search to the second input tensor and accumulator introduce performance problems. Similar to SpGEMM, SpTC has indirect memory accesses to the second input tensor, caused by the non-zero indices of the first input tensor. Take an SpGEMM $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ as an example. A non-zero $\mathbf{A}(0, 1)$ gets, e.g. $\mathbf{B}(1, 1)$, to perform multiplication; while $\mathbf{A}(0, 10)$ computes with, e.g. $\mathbf{B}(10, 2)$. Those irregular memory accesses of $\mathbf{B}$ and the sparse accumulator, which happen more often with the high-dimensional tensors, are not cache friendly. In addition, index search and accumulator, which is used to address irregular memory accesses in SpTC, is more expensive than that in SpGEMM. Our evaluation shows that they takes 54% of SpTC performance on average.

Third, massive memory consumption caused by large input and output tensors and intermediate results creates pressure on the traditional DRAM-based machine. Sparse tensors from real-world applications easily consume a few to dozens of GB memory, while the output tensor could be even larger, because it contains more non-zero elements than any of the input sparse tensor. The intermediate results could be large as well, especially for multi-threading environment where each thread has its own intermediate results. Compared to the well-studied sparse tensor times dense matrices/vectors [29, 37, 39, 64], SpTC results in substantial memory consumption easily, which can be beyond typical DRAM capacity (up to a few hundreds of GB) on a single machine. However, expanding DRAM capacity is not cost effective, while adding cheap but much slower SSD causes significant

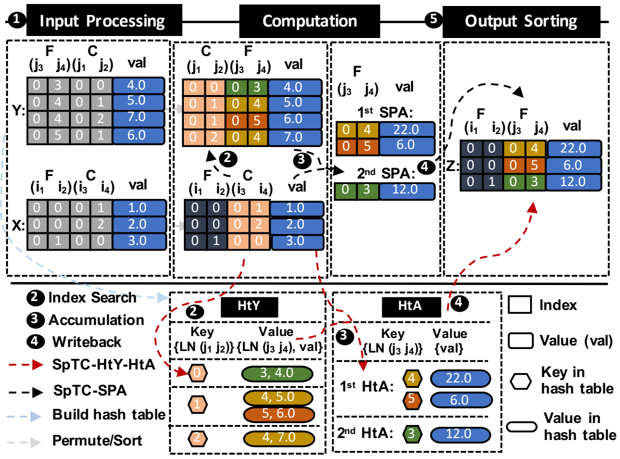

Figure 1. Workflow of the traditional SpTC-SPA and Sparta on $\mathcal{Z} = \mathcal{X} \times^{\{1,2\}}_{\{3,4\}} \mathcal{Y}$.

performance drop. This memory capacity problem is becoming more serious in those HPC applications with increasing dimension size in tensors [7, 13, 18, 20, 47, 55, 66].

To address the first two challenges, we propose Sparta (Algorithm 2) with performance optimizations executed in five stages: input processing, index search, accumulation, writeback, and output sorting. In particular, we employ dynamic arrays to accurately allocate memory space for the accumulator and output tensor to avoid the unknown output challenge. For multi-threading environment, we introduce a thread-private, dynamic object to store the output tensor from each thread for better parallelization. To address the irregular memory access challenge, we perform permutation and sorting on input sparse tensors before computation, thus significantly improve temporary locality of non-zeros in the first input tensor and spacial locality of non-zeros in the second input tensor. Furthermore, we adopt hash table-based approaches based on a large-number representation for the second tensor and accumulator to significantly speed up the process of multi-dimensional search in SpTC. With the above optimizations, Sparta substantially outperforms the traditional SpTC algorithm extended from SpGEMM. By evaluating real data from quantum chemistry and physics, our element-wise Sparta beats their block-sparse algorithms by 7.1× on average.

To address the third challenge, we explore the emerging persistent memory-based heterogeneous memory (HM). In particular, recent Intel Optane DC Persistent Memory Module (PMM) provides bandwidth and latency only slightly inferior to that of DRAM but with only half of the price. PMM often pairs with a small DRAM to build HM, where frequently accessed data objects placed in DRAM and the rest resided in PMM with several TBs of large memory capacity. It is performance-critical to decide the placement of data objects of SpTC (input and output tensors and intermediate results) on PMM-based HM, to make best use of DRAM's high bandwidth and low latency without causing frequent data movement between PMM and DRAM. We first characterize memory read/write patterns associated with those data objects in SpTC, and reveal the performance sensitivity of SpTC to the placement of those data objects on PMM and DRAM. Sparta then prioritizes the data placement between DRAM and PMM statically based on our knowledge on the SpTC algorithm and characterization of data objects for best performance. Sparta effectively avoids unnecessary data movement suffered in the traditional application-agnostic solutions (such as hardware-managed DRAM caching [57, 70, 76] or software-based page hotness tracking [4, 16, 26, 27, 71–73, 77]).

Our main contributions are summarized as follows:

- We introduce the first, high-performance SpTC system for arbitrary-order element-wise sparse tensor contraction, named Sparta. Its implementation will be open-sourced [1]. (Section 3)
- We explore the emerging PMM-based HM to address memory capacity limitation suffered in the traditional tensor computations. (Section 4)
- Evaluating with 15 datasets, Sparta brings $28 - 576\times$ speedup over the traditional SpTC with SPA. With our proposed algorithm- and memory heterogeneity-aware data management, Sparta brings extra performance improvement on HM built with DRAM and PMM over a state-of-the-art software-based data management solution, a hardware-based data management solution, and PMM-only by 30.7% (up to 98.5%), 10.7% (up to 28.3%) and 17% (up to 65.1%) respectively (Section 5).

2 Background

2.1 Sparse Tensors

A tensor can be regarded as a multidimensional array. Each of its dimensions is called a *mode*, and the number of dimensions or modes is its *order*. For example, a matrix of order 2 means it has two modes (rows and columns). We represent tensors with calligraphic capital letters, e.g., $\mathcal{X} \in \mathbb{R}^{I\times J\times K\times L}$ (a tensor with four modes), and x_{ijkl} is its (i, j, k, l)-element. Table 1 summarizes notation and symbols for tensors.

Sparse data, with most of its elements are zeros, is common in various applications. Hence, compressed representations of the sparse tensor have been proposed to save its storage space. In this work, we employ the most common representation, coordinate (COO) format, which is used in Tensor Toolbox [10] and TensorLab [68] (Refer to Section 3.2 for more reasons). A nonzero element is stored as a tuple for its indices, e.g., (i, j, k, l) for a fourth-order tensor, in a two-level pointer array *inds*, along with its nonzero value in a one-dimensional array *val*.

[1] Hide the code address to comply with the double-blind policy.

Table 1. List of symbols and notation.

Symbols	Description
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	Sparse tensors
$\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}$	Tensor contraction between two tensors
N_X	Tensor order of $\mathcal{X}$
I, J, K, L, I_n	Tensor mode sizes
nnz_X	#Non-zeros of the input tensor $\mathcal{X}$
N_F	#Mode-F^X sub-tensors of $\mathcal{X}$
nnz_F	The #Non-zeros of sub-tensors of $\mathcal{X}$
ptr_F	Pointers for mode-F^X sub-tensor locations of $\mathcal{X}$
C^X	A set of contract modes in $\mathcal{X}$, $\{n\}$ in $\times_{\{n\}}^{\{m\}}$ contraction
F^X	A set of free modes in $\mathcal{X}$, $\lvert F^X \rvert + \lvert C^X \rvert = N_X$
C_{nz}^X	Contract mode indices of a non-zero element in $\mathcal{X}$
F_{nz}^X	Free mode indices of a non-zero element in $\mathcal{X}$
val^X	A set of non-zero values in $\mathcal{X}$
val_{nz}^X	Value of a non-zero element in $\mathcal{X}$

2.2 Sparse Tensor Contraction

Tensor contraction, a.k.a. Tensor-Times-Tensor (TTT) or mode-($\{n\}, \{m\}$) product [13], is an extension of matrix multiplication, denoted by

$$\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}, \tag{1}$$

where $\{n\}$ and $\{m\}$ are tensor modes to do contraction.

Example: $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$. This contraction operates on I_3 and I_4 in $\mathcal{X}$ and J_1 and J_2 in $\mathcal{Y}$ ($I_3 = J_1$) and ($I_4 = J_2$). All of the four modes are contract modes (annotated with $C_X = \{3, 4\}$ and $C_Y = \{1, 2\}$), and the other modes are free modes. This example's operation is formally defined as:

$$z_{i_1 i_2 j_3 j_4} = \sum_{i_3 (j_1)=1}^{I_3 (J_1)} \sum_{i_4 (j_2)=1}^{I_4 (J_2)} x_{i_1 i_2 i_3 i_4} x_{j_1 j_2 j_3 j_4}. \tag{2}$$

The number of modes of the output $\mathcal{Z}$, $N_Z = |F_X| + |F_Y| = (N_X - |C_X|) + (N_Y - |C_Y|)$. This is our walk-through example in the following discussion.

2.3 Intel Optane DC Persistent Memory Module

The recent release of the Intel PMM marks the first mass production of byte-addressable NVM. PMM can be configured in *Memory* or *AppDirect* mode. In Memory mode, DRAM becomes a hardware-managed direct-mapped write-back cache to PMM and is transparent to applications. In AppDirect mode, the programmer can explicitly control over the placement of data objects on PMM and DRAM. Sparta works on AppDirect mode and performs better than Memory mode.

PMM brings up to 6TB memory capacity on a single machine with lower latency and bandwidth than DRAM. The read latency of PMM is 174 ns and 304 ns for sequential and random reads respectively, while the counterpart read latency of DRAM is 79 ns and 87 ns. The write latency of PMM is 104 ns and 127 ns for sequential and random writes respectively, while 86 ns and 87 ns for DRAM. In our evaluation platform (Section 5.1), the PMM bandwidth is 39 GB/s and 13 GB/s for read and write respectively, while 104 GB/s and 80 GB/s for DRAM.

3 Sparse Tensor Contraction Algorithm

This section introduces our SpTC algorithms, *SpTC-SPA* and Sparta, to address the challenges of unknown output and irregular memory accesses along with multi-dimensional index search.

3.1 Overview

Figure 1 depicts the workflow of our SpTC algorithm. Our algorithm has five stages: ❶ input processing, ❷ index search, ❸ accumulation, ❹ writeback, and ❺ output sorting, where ❶ and ❺ are called *input/output processing* collectively, and ❷, ❸ and ❹ are *computation* collectively. We describe the *input/output processing* stages in this section and the *computation* stages will be illustrated in Sections 3.2 to 3.5.

Input processing ❶. Figure 1 uses two tiny sparse tensors $\mathcal{X}$ and $\mathcal{Y}$ as input examples. When the modes of $\mathcal{X}$ or $\mathcal{Y}$ are not in the "correct mode order", permutation and sorting are needed. "Correct mode order" means: The contract modes C_X ((i_3, i_4) in Figure 1) are the rightmost modes of $\mathcal{X}$ and C_Y ((j_1, j_2)) are the leftmost modes of $\mathcal{Y}$. $\mathcal{X}$ is first permuted to the “correct mode order“ by exchanging mode indices, which is cheap for COO format [2]. Then according to the new mode order, all the non-zero elements of $\mathcal{X}$ are sorted using a quick sort algorithm with the complexity of $O(nnz_X log(nnz_X)$ where nnz_X is the number of non-zero elements in $\mathcal{X}$. In Figure 1, $\mathcal{X}$ only needs sorting due to its correct mode order; permutation and sorting are both needed for $\mathcal{Y}$. Permutation and sorting are necessary to improve data locality for an efficient implementation of our SpTC algorithms.

Output sorting ❺. The output $\mathcal{Z}$ is not sorted from our algorithms' computation pattern (see Sections 3.2 and 3.5 for details). Depending on the needs, sorting could be acted on $\mathcal{Z}$ after the computation, using the quick sort algorithm. This could avoid its potential sorting when used as an input for the subsequent SpTC computations. In our algorithms, sorting on $\mathcal{Z}$ is by default.

3.2 Sparse Accumulator for High-order Sparse Tensors

Sparse accumulator (SPA) is a popular approach in sparse matrix-sparse matrix multiplication (SpGEMM) [21, 22], which uses a sparse representation to hold the indices and non-zero values of the current active matrix row to do accumulation and is conceptually parallel. We extend SPA to SpTC (named *SpTC-SPA*) for an arbitrary-order sparse tensor and any contraction operation. Figure 1 uses the fourth-order tensor contraction example in Section 2.2 to illustrate the five stages.

Index search ❷. Take $x(0, 1, 0, 0)$ in Figure 1 to illustrate, the indices $(0, 0)$ in mode-3 and 4 are used to search in $\mathcal{Y}$

[2]For example, to exchange modes i_1 and i_2, we only need to switch the pointers of $inds[1]$ and $inds[2]$.

Algorithm 1: *SpTC-SPA*: Sparse tensor contraction of *Example 2:* $\mathcal{Z} = \mathcal{X} \times^{\{1,2\}}_{\{3,4\}} \mathcal{Y}$, extended from SpGEMM [22] with sparse accumulator (SPA)

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$ and $\mathcal{Y} \in \mathbb{R}^{J_1 \times J_2 \times J_3 \times J_4}$, contract modes $C_X = \{3, 4\}$, $C_Y = \{1, 2\}$
Output: The output tensor $\mathcal{Z} \in \mathbb{R}^{I_1 \times I_2 \times J_3 \times J_4}$
1 Permute and sort $\mathcal{X}, \mathcal{Y}$ if not yet;
2 **for** $\mathcal{X}(i_1, i_2, :, :)$ **in** $\mathcal{X}$ **do**
3   Initiate a sparse accumulator *SPA*
4   **for** *Non-zero* $x(i_1, i_2, i_3, i_4)$ **in** $\mathcal{X}(i_1, i_2, :, :)$ **do**
5   **for** *Non-zero* $y(i_3, i_4, j_3, j_4)$ **in** $\mathcal{Y}(i_3, i_4, :, :)$ **do**
6             $v = x(i_1, i_2, i_3, i_4) \times y(i_3, i_4, j_3, j_4)$
7   **if** $SPA(j_3, j_4)$ *exists* **then**
8   Accumulate $SPA(j_3, j_4) += v$
9   **else**
10   Append v to *SPA*
11   Write *SPA* back to $\mathcal{Z}(i_1, i_2, :, :)$
12 Permute and sort $\mathcal{Z}$ as needed
13 **return** $\mathcal{Z}$

for sub-tensor $\mathcal{Y}(0, 0, :, :)$ to multiply with. A linear search iterates non-zeros of $\mathcal{Y}$ until $\mathcal{Y}(0, 0, :, :)$ is found.Similarly in Algorithm 1, we loop all non-zeros of $\mathcal{X}$ by units of sub-tensors in Line 2. For each non-zero $x(i_1, i_2, i_3, i_4)$, we use the indices (i_3, i_4) to do linear search in $\mathcal{Y}$ to locate the sub-tensor $\mathcal{Y}(i_3, i_4, :, :)$ to perform multiplication. The linear search has the complexity $O(nnz_Y)$ by searching all non-zeros of $\mathcal{Y}$ in the worst case. To solve multi-dimensional index search challenge, we will construct $\mathcal{Y}$ as a hash table in Section 3.3.

We explain the reason of using COO format in our algorithms by comparing with the popular compressed storage row (CSR) [69] and its generalization compressed sparse fiber (CSF) [65] formats. For example, we can direct locate row indices in a CSR-represented sparse matrix, but not column indices. Similarly, except the first mode, all the other contract modes have to do linear search as well in a CSF-represented sparse tensor. (Refer to [64, 65] for more details) Thus, index search on CSF-represented $\mathcal{Y}$ will not be significantly better than its COO representation.

Accumulation ❸. In Figure 1, if $y(0, 0, :, :)$ is found, $x(0, 1, 0, 0)$ times every non-zero in $\mathcal{Y}(0, 0, :, :)$, and accumulate the result to *SPA*. For example, $z(0, 1, 0, 3)$ accumulates the product of $x(0, 1, 0, 0)$ and $y(0, 0, 0, 3)$. If $SPA(0, 3)$ is already exists, it adds this product; otherwise, the product along with its indices $(0, 3)$ are appended to the *SPA*.

In Algorithm 1, since every $\mathcal{X}(i_1, i_2, :, :)$ independently accumulates to $\mathcal{Z}(i_1, i_2, :, :)$, the sparse accumulator *SPA* is allocated for each sub-tensor of $\mathcal{X}$. For each non-zero $x(i_1, i_2, i_3, i_4)$, if found $\mathcal{Y}(i_3, i_4, :, :)$ in index search, all non-zeros in $\mathcal{Y}(i_3, i_4, :, :)$ are stored contiguously and have spacial data locality due to the permutation and sorting of $\mathcal{Y}$ in input processing. Since every non-zero in $\mathcal{Y}(i_3, i_4, :, :)$ compute with $x(i_1, i_2, i_3, i_4)$, thus $\mathcal{X}$ gets temporary data locality. If $SPA(j_3, j_4)$ already exists, it adds up the product v; otherwise, v along with its indices (j_3, j_4) are dynamically appended to *SPA*. We also employ the linear search to locate $SPA(j_3, j_4)$ with the complexity $O(|SPA|)$, the size of *SPA*. Once the traverse of all non-zeros in $\mathcal{X}(i_1, i_2, :, :)$ is done, *SPA* contains the final results of $\mathcal{Z}(i_1, i_2, :, :)$. The same multi-dimensional search challenge occurs in index search stage, which is optimized with hash table in Section 3.4.

Writeback ❹. Figure 1 shows the simple write-back stage by copying *SPA* values to $\mathcal{Z}(0, 1, :, :)$. In Section 3.5, we will introduce another temporary data for better parallelization and memory locality.

To solve the challenge of the unknown output size, traditionally two approaches, a two-phase method with symbolic and numeric phases [49] and a loose upper-bound size prediction [5, 14], have been investigated. Symbolic phase counts the number of non-zero elements of the output, which is expensive, then precise memory space is allocated to proceed the computation (numeric phase). A loose upper-bound size prediction uses probabilistic or upper bound methods to allocate large enough memory, which is more than sufficient, for the output. In *SpTC-SPA*, we use dynamic vectors for the *SPA* and output tensor, like progressive method [21] but more precise. The total time complexity of *SpTC-SPA* is

$$
\begin{aligned}
T_{SPA} &= O(nnz_X log(nnz_X) + nnz_Y log(nnz_Y)) \\
&+ O(2 \times nnz_X \times nnz_Y + nnz_Z) + O(nnz_Z log(nnz_Z))
\end{aligned}
\tag{3}
$$

, where the three terms correspond to the time complexity of input processing, *computation* with index search, accumulation, and writeback, and output sorting. Figure 2 illustrates the execution time breakdown of the stages of *SpTC-SPA* (Refer to x-axis meanings in Section 5 and Table 3). This evaluation matches theoretical analysis in Eq. (3), the SpTC time is dominated by computation stage. Stages ❶ and ❺, shown together as *input/output processing*, takes less than 1% of the algorithm. Compared to the two-phase method, our *SpTC-SPA* approach highly reduces the input processing time; while compared to the prediction methods, *SpTC-SPA* can highly reduce *SPA* and the output space. Thus, our *SpTC-SPA* algorithm is a good baseline for SpTCs, by following the spirit of SpGEMM SPA approach with dynamic, precise memory allocation, and good data locality, to support arbitrary-order sparse tensors and any tensor contraction operations. Stages ❷ and ❸ are the performance bottlenecks in Figure 2 for all of our test cases and from Equation (3). We will emphasize optimizing these two stages in Sections 3.2 to 3.5.

3.3 Hash table-represented Sparse Tensor

To address the problems of multi-dimensional index search and inherit good data locality from *SpTC-SPA*, we propose

Algorithm 2: *Sparta*: Sparta sparse tensor contraction for arbitrary-order data.

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_{N_X}}$ and $\mathcal{Y} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_Y}}$, contract modes C_X, C_Y
Output: The output tensor $\mathcal{Z}$
1 Permute and sort $\mathcal{X}$ if needed;
2 Obtain N_F, $|F^X|$, sub-tensors of $\mathcal{X}$, and its ptr_F;
3 Convert $\mathcal{Y}$ to HtY with $LN(C^Y)$ as keys and $(LN(F^Y), val^Y)$ as values;
4 // Compute: $\mathcal{Z} = \mathcal{X} \times_{C_X}^{C_Y} \mathcal{Y}$
5 **for** f **in** $1, \ldots, N_F$ **do**
6   Initiate thread-local HtA with F^Y as keys
7   **for** nz **in** $ptr_F[f], \ldots, ptr_F[f+1]$ **do**
8   **if** $LN(C^X_{nz})$ *is not found in* HtY **then**
9   continue
10   **for** $(LN(F^Y_{nz}), val^Y_{nz})$ **in** $(LN(F^Y), V^Y)$ *of* HtY **do**
11             $v = val^X_{nz} * val^Y_{nz}$
12   **if** $LN(F^Y_{nz})$ *is found in* HtA **then**
13   Accumulate $val^{HT}_{nz} += v$
14   **else**
15   Insert $(LN(F^Y_{nz}), v)$ to HtA
16   Form (F^X_{nz}, F^Y_{nz}) as coordinates and val^{HT}_{nz} as non-zero value and append to $\mathcal{Z}_{local}$
17 Gather thread-local $\mathcal{Z}_{local}$ independently to $\mathcal{Z}$
18 Permute and sort $\mathcal{Z}$ if needed
19 **return** $\mathcal{Z}$

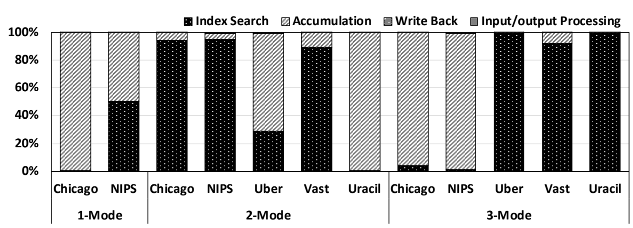

Figure 2. Percentage of execution time breakdown of *SpTC-SPA* (Algorithm 1).

the hash table-represented input tensor $\mathcal{Y}$ with specifications for sparse tensors for the *index search* stage.

Figure 1 depicts the process of converting $\mathcal{Y}$ represented in COO format into a hash table HtY with a large-number representation and its usage in the example SpTC. The index search for $\mathcal{Y}(0, 0, :, :)$ uses $\mathcal{X}$'s contract indices $(0, 0)$, which is taken as the keys of HtY naturally. Since we need to keep the information of free indices of $\mathcal{Y}$, $(0, 3)$, and non-zero values 4.0 for the next stage ❸, the tuple $((0, 3), 4.0)$ is put to the values of HtY. Since the keys of HtY are index tuples, as the tensor order grows, it is difficult and time-consuming to do key matching on multi-dimensional tuples. We introduce a *large-number representation*, noted as the LN function in Figure 1, which converts a sparse index tuple to a large index in a dense pattern. For example, $(0, 3)$ tuple is converted to $3 = 0 \times J_4 + 3$. Unique identifiers are extremely important for a fast hash table search. This large-number representation obtains unique numbers for every tuple of keys in HtY, hence the index search becomes faster on HtY by doing integer comparison for keys. To create HtY from $\mathcal{Y}$ in COO format, we use separate chaining hash table [1] with given-sized buckets to distribute the keys. Compared to COO format, the contract indices have no duplication due to the unique key feature of a hash table, which reduces the index search space. To maintain the good spacial data locality from Algorithm 1, for the non-zeros having the same key in $\mathcal{Y}$, we adopt dynamic array to construct the values of HtY.

The creation and usage of HtY for an arbitrary-order SpTC with random contract modes C_X and C_Y are illustrated in Algorithm 2. The three for-loops are in the same order with those in Algorithm 1. The first and second loop sub-tensors in $\mathcal{X}$ and non-zeros in the sub-tensor using ptr_F to indicate locations respectively. The indices of contract modes C_Y and the tuple of free modes and non-zero value (F^Y, val^Y) are taken as the keys and values of HtY respectively in Line 3. For each non-zero element nz, we search $LN(C^X_{nz})$, the large-number representation of the contract indices C_X of $\mathcal{X}$, in HtY (Line 8). Compared to the linear search in *SpTC-SPA* with the complexity $O(nnz_Y)$, the time complexity of hash table search on HtY is significantly reduced to $O(1)$ [1]. We also optimize input processing, the COO-to-hashtable conversion is faster than permutation and sorting of $\mathcal{Y}$, $O(nnz_Y)$ versus $O(nnz_Y log(nnz_Y))$.

Our proposed hash table-represented sparse tensor with the large-number compressed keys highly improves the SpTC performance by efficiently addressing multi-dimensional index search issue and maintain temporary and spacial data locality. To reduce the frequency of index search, we always treat the larger input tensor as $\mathcal{Y}$ in our SpTC algorithms.

3.4 Hash table-based Sparse Accumulator

Hash table [6, 49–51], hashmap [15], and heap [9] are popular data structures to represent the accumulator in state-of-the-art SpGEMM research, where hash table performs the best from prior evaluations [49]. As mentioned in Section 3.2 and Figure 2, stage ❸ in *SpTC-SPA* could dominate the performance of an SpTC. To more efficiently accumulate the intermediate results, we propose a hash table-based accumulator HtA, shown in Figure 1. We take the free indices of $\mathcal{Y}$, $(0, 3)$, as a key and refer to the intermediate result as the values of the hash table. Separate chaining hash table and the large-number representation LN are also adopted here for fast key matching and hash search.

We observe the key of HtA ($(0, 3)$ in Figure 1) is the same with the free indices of $\mathcal{Y}$ in the value tuples of HtY (also $(0, 3)$). To avoid the key conversion for HtA, we convert the free indices of $\mathcal{Y}$ to the large-number representation in stage ❶ (Line 3 in Algorithm 2). We directly retrieve the keys from the values of HtY, avoiding mode indices-key conversion between HtY and the accumulator HtA during computation. As depicted in Figure 1 and Algorithm 2, the accumulation performs similar to *SpTC-SPA* but on hash table HtA instead.

By far, we form the Sparta SpTC algorithm (Algorithm 2). Compared to *SpTC-SPA*, we replace $\mathcal{Y}$ and SPA with two hash table HtY and HtA with large-number representation respectively. Sparta solves the multi-dimensional index search challenge (Section 1), get faster processing for input $\mathcal{Y}$, extract unnecessary index computation/conversion out of the computation, while maintain the good data locality shown in *SpTC-SPA*, to reduce the SpTC execution time. The total time complexity of Sparta:

$$
\begin{aligned}
T_{Sparta} &= O(nnz_X log(nnz_X) + nnz_Y) \\
&+ O\big(2 \times nnz_X \times nnz_{F_{avg}} + nnz_Z\big) + O(nnz_Z log(nnz_Z))
\end{aligned}
\tag{4}
$$

, where $nnz_{F_{avg}}$ is the average size of all sub-tensors (e.g. $\mathcal{Y}(j_1, j_2, :, :)$ in Algorithm 1). The three terms correspond to the time complexity of stages ❶, *computation* with ❷, ❸, and ❹, and ❺. Eq. (4) shows that depending on different sparse tensors, the SpTC time could be dominated by different stages (more details in Section 5).

3.5 Parallelization

We parallelize all the five stages of *SpTC-SPA* and Sparta algorithms. For stage ❶, since permutation takes negligible time, we parallelize the quick sort algorithm using OpenMP tasks, which is also used in stage ❺. Sparta has the COO-to-hashtable representation for $\mathcal{Y}$ in stage ❶, we parallelize subtensors of $\mathcal{Y}$ and use locks on the buckets of HtY to ensure correct insertion and updates. Since the separate chaining hash table relatively evenly distributes the search requests, locks on multi-threading gets an acceptable performance (7.8× speedup on average over a sequential version using 12 threads in our experiments).

In computation, we parallelize the outermost loop for subtensors of $\mathcal{X}$ (Line 2 in Algorithm 1 and Line 5 in Algorithm 2). Thus, the sparse accumulator SPA in *SpTC-SPA* and hash table accumulator HtA in Sparta are both thread-private and each thread can do accumulation independently. Due to the dynamic output structure, directly write the intermediate thread-local SPA or HtA results to $\mathcal{Z}$ is not feasible. We introduce thread-local dynamic Z_{local} in Algorithm 2 to write the intermediate results. After one thread completes its execution, we have the size of Z_{local} which can be used to allocate the space for $\mathcal{Z}$. Then each thread could writes its Z_{local} to $\mathcal{Z}$ in a parallel pattern. The introduction of Z_{local} helps to solve the unknown output challenge (Section 1) in multi-threading parallel environment and improves stage ❹ with the cost of the affordable thread-local storage Z_{local}.

4 Data Placement on PMM-based Heterogeneous Memory Systems

We discuss our approaches to leveraging HM to address the memory capacity bottleneck of SpTC.

4.1 Characterization Study

To motivate our solution of data placement on heterogeneous memory, we characterize memory accesses of major data objects of Sparta (Algorithm 2), in terms of access patterns (sequential/random and read/write) in Table 2. Five stages, input processing ❶, computation (combined ❷ index search, ❸ accumulation, ❹ writeback) and output sorting ❺, are considered with six major data objects, i.e., the two input tensors ($\mathcal{X}$ and $\mathcal{Y}$), the hash table-represented second input tensor (HtY), thread-local hash table-based accumulator (HtA), the thread-local temporary data ($\mathcal{Z}_{local}$), and the output tensor ($\mathcal{Z}$).

We study the performance impact of the placement of six data objects on tensor Nell-2 with 2-Mode contraction in Figure 3, by evaluating Sparta on a server with an HM with PMM and DDR4 (described in Section 5.1). We use the execution time to reflect the underneath PMM and DRAM memory characteristics and an accurate performance behavior of SpTC. Our baseline is the Sparta execution time when residing all data in DRAM, which achieves the fastest performance on the HM. We perform six tests: each one by placing only one data object in PMM, while leaving the others stay in DRAM. We have three interesting observations that guide our data placement for Sparta.

Observation 1: Performance difference between read and write matters a lot to performance of Sparta. For example, the memory access pattern associated with $\mathcal{Y}$ in the stage ❶ is sequential read-only, and placing it on PMM causes ignorable performance loss; In contrast, the memory access pattern associated with $\mathcal{Z}_{local}$ in the stage ❸ is sequential write-only, and placing it on PMM causes 12.9% performance loss. The bandwidth difference between read and write on PMM is about 3×, which leads to the difference in Sparta's performance.

Observation 2: Sequential and random accesses have large performance difference. For example, the memory access pattern associated with $\mathcal{Y}$ in the stage ❶ is sequential read-only, and placing it on PMM causes ignorable performance loss; In contrast, the memory access pattern associated with HtY in the stage ❷ is random read-only, and placing it on PMM causes 30.8% performance loss. The performance difference between sequential and random accesses on PMM is due to the unique architecture of PMM (e.g., the combining buffer in devices [75]); Sequential accesses also makes hardware prefetching more effective for improving data locality.

Observation 3: The performance of Sparta is not sensitive to the placement of some data objects on PMM. For

Table 2. Memory access patterns associated with data objects in six stages ("Ran" = Random; "Seq" = Sequential; "RW" = Read-Write; "RO" = Read-Only; "WO" = Write-Only). (Note that for Ht$\mathfrak{Z}$ in Summation, we employ temporal locality to always maintain the most frequently used bucket in DRAM. The memory access pattern of key-value nodes within the bucket is still random. For $\mathfrak{Z}_{input}$, the first two passes are "Seq, RO" and the last pass is "Ran, RO".)

Stages	Data Objects							
	$\mathfrak{X}$	$\mathfrak{Y}$	HtY	HtA	$\mathfrak{Z}_{local}$	$\mathfrak{Z}$	$\mathfrak{Z}_{input}$	Ht$\mathfrak{Z}$
Input Processing ❶	Ran, RW	Seq, RO	Ran, RW	-	-	-	-	-
Index Search ❷	Seq, RO	-	Ran, RO	-	-	-	-	-
Accumulation ❸	-	-	-	Ran, RW	Seq, WO	-	-	-
Writeback ❹	-	-	-	-	Seq, RO	Seq, WO	-	-
Output Sorting ❺	-	-	-	-	-	Ran, RW	-	-
Summation ❻	-	-	-	-	-	Seq, RO	Ran, RO	Ran, RW

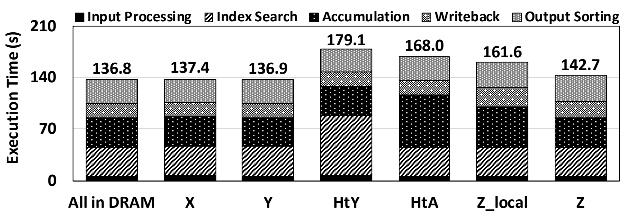

Figure 3. Performance after placing a data object in PMM while leaving others in DRAM. The x axis shows the data object placed in PMM. "All in DRAM" means all data objects are placed in DRAM.

exampling, placing $\mathfrak{X}$ and $\mathfrak{Y}$ on PMM, Sparta has ignorable performance loss, because of the memory access patterns discussed in the above two observations.

The first two observations are unique to PMM compared to traditional DRAM. Read and write, sequential and random accesses both has small performance difference in DRAM. We get the same observations for other 14 datasets.

4.2 Data Placement Strategy

Driven by the characterization results, we use the following data placement strategy. $\mathfrak{X}$ and $\mathfrak{Y}$ is always on PMM, because of the observation 3. For the other four data objects, we decide their placement in DRAM, following the priority of $HtY > HtA > \mathfrak{Z}_{local} > \mathfrak{Z}$. For each of the four data objects, we make best efforts to place it into DRAM. This means that given a data object, if there is remaining DRAM space after excluding that consumed by the data objects with higher priority, then that object is placed into DRAM as much as possible; If there is no remaining DRAM space, that object is placed into PMM.

To implement the above data placement strategy, we must estimate the memory consumption of the four data objects, to decide whether they should be placed into DRAM or not. We discuss it as follows.

The placement of HtY**.** We estimate the memory consumption of HtY using Equation 5 based on tensor information and knowledge on data structures used in HtY. In Equation 5, $Size_{HtY}$ is the memory consumption of HtY; $Size_{ep}$, $Size_{idx}$ and $Size_{val}$ are the size of the entry pointer for a bucket in HtY, the size of an index, and the size of a value, respectively; $\#Buckets_{HtY}$ is the number of buckets in HtY; $nnz_{\mathfrak{Y}}$ is the number of non-zero elements in $\mathfrak{Y}$; N_Y is the number of modes of $\mathfrak{Y}$.

$$Size_{HtY} = Size_{ep} \cdot \#Buckets_{HtY} + nnz_{\mathfrak{Y}} \cdot (Size_{idx} \cdot N_Y + Size_{val} + Size_{ep}) \quad (5)$$

Equation 5 includes the memory consumption for metadata (i.e., the pointers pointing to each bucket in the hash table, modeled as $Size_{ep} \cdot \#Buckets_{HtY}$); Equation 5 also includes the memory consumption for storing all non-zero elements of $\mathfrak{Y}$ in HtY, each of which consumes memory for an index, a value, and a pointer pointing to another element, modeled as $Size_{idx} \cdot N_Y + Size_{val} + Size_{ep}$.

To use Equation 5, we must know $nnz_{\mathfrak{Y}}$ and $\#Buckets_{HtY}$. $nnz_{\mathfrak{Y}}$ as a tensor feature is typically known; $\#Buckets_{HtY}$ is defined by the user, and hence is known.

The placement of HtA**.** We use Equation 6 to estimate the memory consumption of HtA. While Equation 5 estimates the exact memory consumption, Equation 6 gives an upper bound on the memory consumption ($Size_{HtA}$).

$$Size_{HtA} = Size_{ep} \cdot \#Buckets_{HtA} + nnz^X_{Fmax} \cdot nnz^Y_{Fmax} \cdot (Size_{idx} \cdot |F^Y| + Size_{val} + Size_{ep}) \quad (6)$$

In Equation 6, $|F^Y|$ is the number of free modes of $\mathfrak{Y}$; nnz^X_{Fmax} is the maximum size of all non-zero sub-tensors $\mathfrak{X}(F^X, :, \ldots, :)$; nnz^Y_{Fmax} represents the maximum size of all non-zero sub-tensors $\mathfrak{Y}(C^Y, :, \ldots, :)$. The product of nnz^X_{Fmax} and nnz^Y_{Fmax} gives an upper bound on the number of non-zero elements stored in HtA.

Equation 6 gives an upper bound, because we do not know the exact number of non-zero elements in $\mathfrak{Y}$ that have the same contract indices as those in $\mathfrak{X}$; We use the maximum number to give an upper bound and ensure there is enough space allocated in DRAM for HtA. Using the upper bound does not cause significant waste of DRAM space, because HtA per thread is usually 10-50 MB (even with the largest dataset using 768GB memory in our evaluation). Given tens of threads in a machine, the upper bound takes only a few GB of DRAM, which is typically a small portion of DRAM space in an HPC server.

To use Equation 6, we must know nnz^X_{Fmax} and nnz^Y_{Fmax}. nnz^X_{Fmax} and nnz^Y_{Fmax} are known after the stage ❶, and the dynamic allocation of HtA can happen after the stage ❶ but before the stage ❷ where HtA is accessed. Hence, Equation 6 can be used to effectively direct data placement. In addition, DRAM is evenly partitioned between threads for placing HtA per thread, in order to avoid load imbalance.

The placement of $\mathfrak{Z}_{local}$**.** The memory consumption of $\mathfrak{Z}_{local}$ can be estimated after HtA is filled (Line 16 in Algorithm 2) and before memory allocation for $\mathfrak{Z}_{local}$ happens.

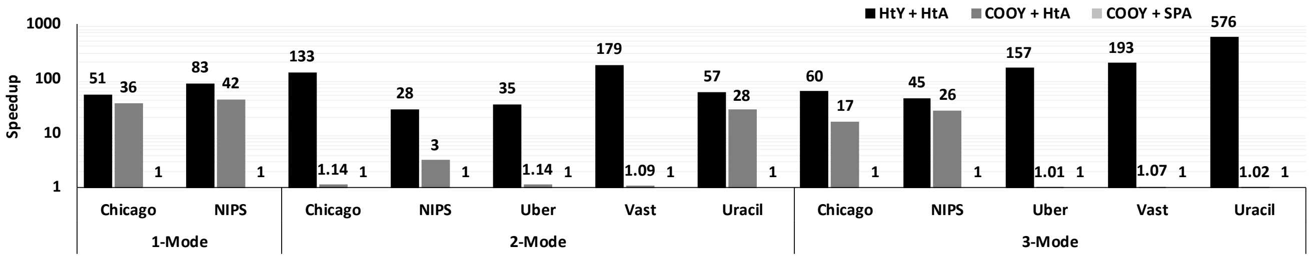

Figure 4. Speedups of HtY+HtA (i.e., Sparta) and COOY+HtA over COOY+SPA (i.e., SpTC-SPA) for SpTCs on Chicago, NIPS, Uber, Vast and Uracil with 1-mode, 2-mode and 3-mode.

The memory consumption of $\mathcal{Z}_{local}$ is equal to the size of HtA plus the size of $F_{nz}^{X} \cdot nnz_{HtA}$, where F_{nz}^{X} refers to free indices of a non-zero element in $\mathcal{X}$ and nnz_{HtA} is the number of non-zero elements in HtA. In addition, DRAM is evenly partitioned between threads for placing $\mathcal{Z}_{local}$ per thread, in order to avoid load imbalance.

The placement of $\mathcal{Z}$. The size of $\mathcal{Z}$ is the summation of the size of $\mathcal{Z}_{local}$ in each thread. The size of $\mathcal{Z}$ is estimated in Line 17 in Algorithm 2, before memory allocation for $\mathcal{Z}$ happens.

Static placement vs. dynamic migration. The data placement strategy in Sparta is static, which means a data object, once placed in DRAM or PMM, is not migrated to PMM or DRAM in the middle of execution. The traditional solutions are application agnostic and dynamic. They track page (or data) access frequency [4, 16, 26, 27, 71–73, 77] or manage DRAM as a hardware cache for PMM [44, 57, 70, 76] to decide the placement of data objects on DRAM and PMM. The traditional solutions, once determining frequently accessed data (hot data), dynamically migrate hot or cold data between DRAM and PMM for high performance. However, those dynamic migration solutions cannot work well in our case because they can cause unnecessary data movement. For example, the performance of Sparta is not sensitive to the placement of $\mathcal{X}$ and $\mathcal{Y}$ on PMM and DRAM, because of their sequential read patterns. The dynamic solutions can unnecessarily migrate them to DRAM for high performance. For another example, HtY has a random access pattern. Any dynamic migration solution cannot effectively capture its pattern and hence causes unnecessary data migration. Our evaluation results in Section 5.5 show that two dynamic migration solutions (i.e., hardware-based Memory mode and software-based IAL [74]) perform worse than Sparta by 10.7% (up to 28.3%) and 30.7% (up to 98.5%) respectively.

Other datasets. We evaluate 15 datasets in total, and 11 of them shows the same priority for data placement (i.e., $HtY > HtA > \mathcal{Z}_{local} > \mathcal{Z}$). However, there are four cases showing different priority (i.e., $HtA > HtY > \mathcal{Z}_{local}$ and $\mathcal{Z}$). For those uncommon cases, we can use the same methodology to determine data placement; Our methods to determine the sizes of the data objects are still valid.

5 Evaluation

5.1 Evaluation Setup

Platforms. The experiments in Sections 5.2, 5.3 and 5.4 are run on a Linux server consisting of 96 GB DDR4 memory and Intel Xeon Gold 6126 CPU including 12 physical cores with 2.6 GHz frequency on one socket. The experiments in Section 5.5 are run on an Intel Optane Linux server containing Intel Xeon Cascade-Lake CPU including 24 physical cores with 2.3 GHz frequency. Each socket has 6×16 GB of DRAM and 6×128 GB Intel Optane DIMMs. All implementations (Sparta and other approaches) are compiled by gcc-7.5 and OpenMP 4.5 with -O3 optimization option. All experiments were conducted on a single socket with one thread per physical core. Each workload was run 10 times and we report the average execution time.

Datasets and expression. We use sparse tensors, derived from real-world applications, that appear in Table 3, ordered by modes and nonzero density. The tensors are included in FROSTT [62]. Tensor Uracil [7, 17] is from a real-world CCSD model in quantum chemistry, formed by cutting off values smaller than 1×10^{-8} verified by chemists.

For some SpTC, the memory requirement is larger than the system memory capacity. We do not evaluate the performance of those SpTC. For a tensor with different expression, we use a "*" to distinguish. For example, Chicago and Chicago* are the same tensors with different expression. Sparta includes five stages, ❶ input processing, computation (combined ❷, ❸, ❹), and ❺ output sorting.

5.2 Overall Performance

Figure 4 shows the performance comparison of using HtY+HtA (i.e., Sparta), COOY+HtA and COOY+SPA (i.e.,

Table 3. Characteristics of sparse tensors in the evaluation.

Tensors	Order	Dimensions	#Nonzeros	Density
Nell-2	3	$12K \times 9K \times 28K$	76M	2.4×10^{-5}
NIPS	4	$2K \times 3K \times 14K \times 17K$	3M	1.8×10^{-6}
Uber	4	$183 \times 24 \times 1K \times 1K$	3M	2×10^{-4}
Chicago	4	$6K \times 24 \times 77 \times 32$	5M	1×10^{-2}
Uracil	4	$90 \times 90 \times 174 \times 174$	10M	4.2×10^{-2}
Flickr	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
Delicious	4	$533K \times 17M \times 2M \times 1K$	140M	4.3×10^{-15}
Vast	5	$165K \times 11K \times 2 \times 100 \times 89$	26M	8×10^{-7}

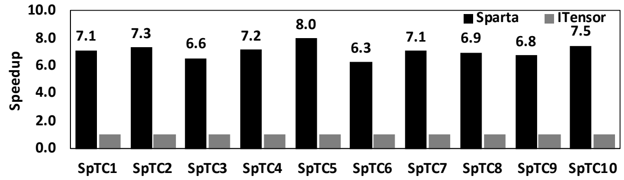

Figure 5. Speedups of Sparta over ITensor on Hubbard-2D model using different SpTC expression with different sparse input tensors.

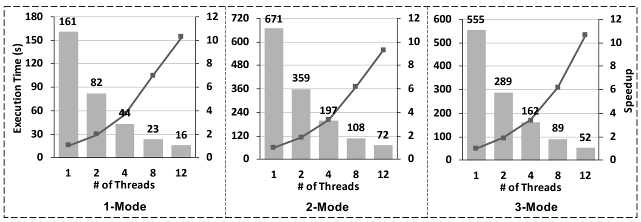

Figure 6. Thread scalability of parallel Sparta on SpTCs on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode.

SpTC-SPA) on tensors Chicago, NIPS, Uber, Vast and Uracll with 1-mode, 2-mode and 3-mode SpTC respectively. In Figure 4, we observe that HtY+HtA significantly outperforms COOY+HtA with 1.4 – 565× performance improvement. The results show that HtY is much efficient than COOY. Also, we found that COOY+HtA significantly outperforms COOY-+SPA with 1% – 42× performance improvement. The results expose that HtA is much efficient than SPA.

We observe that the performance improvement of Sparta over COOY-SPA on Uracil with 3-mode is larger than others. This is because the execution time of stage ❷ dominates the total execution time (99.3%) and the total execution time is relatively large (1072 seconds) than others. Based on the time complexity difference between HtY and COOY in stage ❷, the larger execution time SpTC spends, the larger performance improvement Sparta can achieve. In Figure 2, the total execution is dominated by stages ❷ and ❸ in COOY-SPA (99.6%). Since the execution time of ❷ and ❸ is highly reduced by Sparta, the execution time of stages ❷ and ❸ might not be the bottleneck of an SpTC. In our experiments with Sparta, the time in ❷ accounts for 4.7%; the time of stage ❸ is 61.6%; the time of stage ❹ is 9.6%; the stage ❶ accounts for 3.3% and ❺ is 20.8%.

5.3 Performance Comparison to ITensor

In this experiment, we compare the performance of Sparta and ITensor. ITensor [20] is a state-of-the-art library for multi-threading, block-sparse tensor contraction on a single machine, which is the most related to Sparta among other works. ITensor is configured with its best configurations described in its repository [2]. SpTC expressions with different tensors (SpTC1 to SpTC10) are from a well-known quantum physics model (Hubbard-2D) [19] in ITensor [2], and those tensors are formed by cutting off values smaller than 1×10^{-8} verified by physicists. We choose ITensor as a representative for comparison rather than others (such as libtensor [47], TiledArray [55], CTF [66] and TACO [32]), because libtensor only supports sequential block-wise SpTC [47] while TiledArray and CTF are distributed, and TACO does not support high-order SpTC yet. Figure 5 shows the performance comparison between Sparta and ITensor. We observe that Sparta significantly outperforms ITensor with 7.1× performance improvement on average. We demonstrate that Sparta can also be employed for applications featured with block-wise SpTC.

5.4 Thread Scalability

Figure 6 shows the performance of parallel Sparta over the sequential version. Sparta achieves 10.2×, 9.3× and 10.7× speedup on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode using 12 threads. Different stages have different thread scalability. Evaluation with 15 datasets using Sparta, the average speedup of parallel execution over sequential execution achieves: 10.4 × in stage ❷; 10.9 × in stage ❸; 9.5 × in stage ❹; 6.8 × in stage ❶ and 6.2 × in ❺. Though the thread scalability of stages ❶ and ❺ are not as good as the computation stages (❷,❸,❹), the SpTC is always dominated by the computation stages. Thus, Sparta achieves high overall thread scalability.

5.5 Sparta on Heterogeneous Memory Systems

Now we study the performance of Sparta on HM, compared with a state-of-the-art solution for HM management (i.e., IAL (Improved Active List) [74]), hardware-managed cache approach (i,e, PMM Memory mode), Optane-only (i.e., AppDirect mode with assigning all data objects to Optane) and DRAM-only (i.e., assign all data objects to DRAM). IAL is configured with its best configurations based on the IAL repository [3]. Figure 9 shows the peak memory consumption of SpTCs in the experiment.

As shown in Figure 7, Sparta outperforms IAL with 30.7% performance improvement on average (up to 98.5%). Also, Sparta achieves 10.7% (up to 28.3%) and 17% (up to 65.1%) performance improvement on average than PMM Memory mode and Optane-only approaches respectively. Furthermore, Sparta is comparable to DRAM-only approach with only 6% performance loss. For some SpTC (e.g., Chicago* with 3-mode), because the memory bandwidth requirement is small, the performance difference between Sparta and Optane-only is small. For example, if we assign all data objects to DRAM (i.e., DRAM-only, the configuration with the best performance) on Chicago* with 3-mode, the performance improvement is only 6% over Optane-only.

In Figure 8, we observe that the average PMM memory bandwidth of IAL is larger than Sparta. This is because IAL causes undesirable data movement and such data movement causes higher PMM memory bandwidth. The average DRAM memory bandwidth of PMM memory mode is larger than Sparta because PMM Memory mode manages DRAM as a

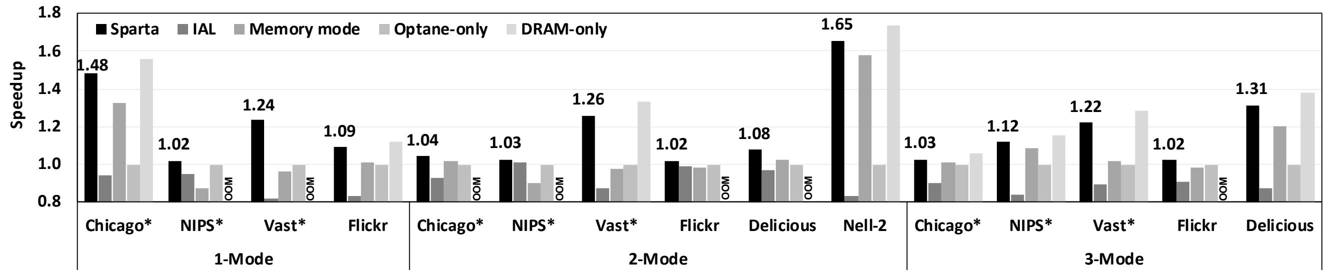

Figure 7. Speedups of Sparta, IAL, Memory mode and Dram-only over Optane-only for SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

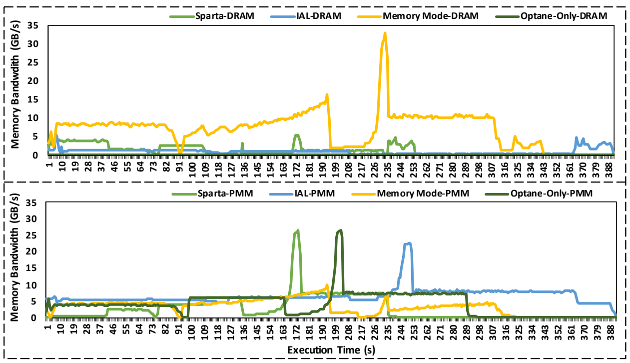

Figure 8. Memory bandwidth of Sparta, IAL, PMM Memory mode and Optane-only on Vast with 1-mode SpTC.

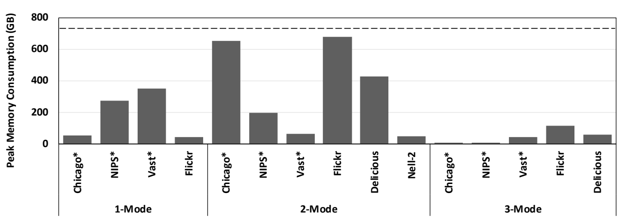

Figure 9. Peak memory consumption of SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

hardware cache for PMM and unnecessarily migrates data objects to DRAM for high performance without being able to be aware of access patterns of data objects.

6 Related Work

Tensor contraction. Tensor contraction has a long history in scientific computing in chemistry, physics, and mechanics. Dense tensor contraction has been studied for decades on diverse hardware platforms [8, 23, 25, 30, 31, 34, 35, 42, 48, 60, 66, 67]. The state-of-the-art sparse tensor contractions emphasize on block-sparse tensor contractions, between two tensors with non-zero dense blocks. The general approaches extract dense block-pairs of the two input tensors, then do multiplication by calling dense BLAS linear algebra and have the output tensor pre-allocated from domain knowledge or a symbolic phase [24, 28, 54, 61**?**], such as libtensor [18, 47], TiledArray [55], and Cyclops Tensor Framework [36]. Our work proposes an efficient element-sparse tensor contraction and shows its performance advantages if a practical cutoff value gets quantum chemistry or physics data below 5% non-zero density. This work will be valuable for deep learning after introducing sparsity from model or data compression.

Sparse tensor formats. Researchers are making continuous effort on developing sparse tensor formats for high-order data, including compressed sparse fiber (CSF) [65], balanced and mixed-mode CSF (BCSF, MM-CSF) [52, 53], flagged COO (F-COO) [43], and hierarchical coordinate (HiCOO) [39] for general sparse tensors, and mode-generic and -specific formats for structured sparse tensors [11]. We choose COO format in this work as a start because CSF format also needs expensive search to locate $\mathcal{Y}$ due to multi-dimensionality. Our hashtable-represented $\mathcal{Y}$ is a new approach to compress a sparse tensor customized to a tensor contraction. This work is orthogonal to the tensor format works and will adopt a more compressed format for the sparse tensor $\mathcal{X}$ according to SpTC operations.

Sparse matrix-matrix multiplication. Sparse matrix-matrix multiplication (SpGEMM) has been well-studied [5, 6, 9, 15, 22, 32, 45, 49–51]. Our hash table implementations can be improved from more advanced algorithms in [6, 46, 50, 51].

Data management on heterogeneous memory systems attracts a lot of attention recently. Many research efforts [4, 16, 26, 27, 71–73, 77] use a software-based solution to track data objects or page hotness to decide data placement on HM; Many research efforts [44, 57, 70, 76] use a hardware-based solution to profile memory accesses and decide data placement on HM. All of those solutions use dynamic migration and are application-agnostic. Sparta is different from them in terms of static data placement and application awareness.

7 Conclusions

SpTC plays an important role in many applications. However, how to efficiently implementing SpTC faces multiple challenges, such as unpredictable output size, time-consuming process to handle irregular memory accesses, and massive memory consumption. In this paper, we introduce Sparta, a high performance SpTC algorithm to address the above challenges based on the innovation of leveraging new data representation, data structures and emerging HM architecture. Sparta shows superior performance: evaluating with 15 datasets, we show that Sparta brings 28 – 576× speedup over the traditional sparse tensor contraction; With our algorithm- and memory heterogeneity-aware data management, Sparta brings extra performance improvement on HM built with DRAM and PMM over a state-of-the-art software-based data management solution, a hardware-based data management solution and PMM-only by 30.7% (up to 98.5%), 10.7% (up to 28.3%) and 17% (up to 65.1%) respectively.

References

[1] Hash table, https://en.wikipedia.org/wiki/Hash_table.

[2] ITensor: A C++ library for efficient tensor network calculations, https://github.com/ITensor/ITensor.

[3] Repository of Nimble Page Management for Tiered Memory Systems in ASPLOS2019, https://github.com/ysarch-lab/nimble_page_management_asplos_2019.

[4] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 631–644, 2017.

[5] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 406–419. Springer, 2010.

[6] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

[7] Edoardo Apra, Eric J Bylaska, Wibe A De Jong, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Marat Valiev, HJJ van Dam, Yuri Alexeev, James Anchell, et al. Nwchem: Past, present, and future. *The Journal of chemical physics*, 152(18):184102, 2020.

[8] Alexander A Auer, Gerald Baumgartner, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[9] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[10] Brett W. Bader, Tamara G. Kolda, et al. Matlab tensor toolbox version 3.1. Available online, June 2019.

[11] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6, Sept 2012.

[12] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Prakash Murali, Shivmaran S. Pandian, Yogish Sabharwal, and Dheeraj Sreedhar. On optimizing distributed Tucker decomposition for sparse tensors. In *Proceedings of the 32nd ACM International Conference on Supercomputing*, ICS '18, 2018.

[13] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014.

[14] Edith Cohen. On optimizing multiplications of sparse matrices. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 219–233. Springer, 1996.

[15] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 693–702. IEEE, 2017.

[16] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *European Conference on Computer Systems*, 2016.

[17] Evgeny Epifanovsky, Karol Kowalski, Peng-Dong Fan, Marat Valiev, Spiridoula Matsika, and Anna I Krylov. On the electronically excited states of uracil. *The Journal of Physical Chemistry A*, 112(40):9983–9992, 2008.

[18] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry*, 34(26):2293–2309, 2013.

[19] Tilman Esslinger. Fermi-hubbard physics with atoms in an optical lattice. 2010.

[20] Matthew Fishman, Steven R White, and E Miles Stoudenmire. The ITensor software library for tensor network calculations. *arXiv preprint arXiv:2007.14822*, 2020.

[21] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[22] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[23] Albert Hartono, Qingda Lu, Thomas Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E Bernholdt, Marcel Nooijen, Russell Pitzer, J Ramanujam, et al. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009.

[24] Thomas Hérault, Yves Robert, George Bosilca, Robert Harrison, Cannada Lewis, and Edward Valeev. *Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure.* PhD thesis, Inria-Research Centre Grenoble–Rhône-Alpes, 2020.

[25] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[26] Takahiro Hirofuchi and Ryousei Takano. Raminate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 112–125, New York, NY, USA, 2016. ACM.

[27] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos — os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, June 2017.

[28] Daniel Kats and Frederick R Manby. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics*, 138(14):144101, 2013.

[29] O. Kaya and B. Uçar. Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018.

[30] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. Optimizing tensor contractions in ccsd (t) for efficient execution on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 96–106, 2018.

[31] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and Ponnuswamy Sadayappan. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95. IEEE, 2019.

[32] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[33] Christoph Koppl and Hans-Joachim Werner. Parallel and low-order scaling implementation of hartree–fock exchange using local density fitting. *Journal of chemical theory and computation*, 12(7):3122–3134, 2016.

[34] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. TensorLy: Tensor learning in Python. *CoRR*, abs/1610.09555, 2018.

[35] Pai-Wei Lai, Kevin Stock, Samyam Rajbhandari, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2013.

[36] Ryan Levy, Edgar Solomonik, and Bryan K Clark. Distributed-memory dmrg via sparse and dense parallel tensor contractions. *arXiv preprint arXiv:2007.05540*, 2020.

[37] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc. Model-driven sparse CP decomposition for higher-order tensors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1048–1057, May 2017.

[38] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, IAˆ3 '16, pages 26–33, Piscataway, NJ, USA, 2016. IEEE Press.

[39] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Dallas, TX, USA, November 2018.

[40] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 227–237, New York, NY, USA, 2019. ACM.

[41] Lingjie Li, Wenjian Yu, and Kim Batselier. Faster tensor train decomposition for sparse data. *arXiv preprint arXiv:1908.02721*, 2019.

[42] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P Sadayappan. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[43] 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)*, pages 47–57, Sept 2017.

[44] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018.

[45] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

[46] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[47] Samuel Manzer, Evgeny Epifanovsky, Anna I Krylov, and Martin Head-Gordon. A general sparse tensor framework for electronic structure theory. *Journal of chemical theory and computation*, 13(3):1108–1116, 2017.

[48] Devin Matthews. High-performance tensor contraction without BLAS. *CoRR*, abs/1607.00291, 2016.

[49] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–10, 2018.

[50] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90:102545, 2019.

[51] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110. IEEE, 2017.

[52] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. An efficient mixed-mode representation of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 49:1–49:25, New York, NY, USA, 2019. ACM.

[53] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P Sadayappan. Load-balanced sparse mttkrp on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 123–133. IEEE, 2019.

[54] David Ozog, Jeff R Hammond, James Dinan, Pavan Balaji, Sameer Shende, and Allen Malony. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *2013 42nd International Conference on Parallel Processing*, pages 30–39. IEEE, 2013.

[55] Chong Peng, Justus A Calvin, Fabijan Pavosevic, Jinmei Zhang, and Edward F Valeev. Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using tiledarray framework. *The Journal of Physical Chemistry A*, 120(51):10231–10244, 2016.

[56] Ioakeim Perros, Evangelos E. Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. SPARTan: Scalable PARAFAC2 for large & sparse data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 375–384, New York, NY, USA, 2017. ACM.

[57] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *International Conference on Supercomputing (ICS)*, May 2011.

[58] Christoph Riplinger, Peter Pinski, Ute Becker, Edward F Valeev, and Frank Neese. Sparse maps—a systematic infrastructure for reduced-scaling electronic structure methods. ii. linear scaling domain based pair natural orbital coupled cluster theory. *The Journal of chemical physics*, 144(2):024109, 2016.

[59] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

[60] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 193–202, Dec 2016.

[61] Ilia Sivkov, Patrick Seewald, Alfio Lazzaro, and Jürg Hutter. DBCSR: A blocked sparse tensor algebra library. *arXiv preprint arXiv:1910.13555*, 2019.

[62] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. Frostt: The formidable repository of open sparse tensors and tools, 2017.

[63] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE, 2016.

[64] Shaden Smith and George Karypis. Accelerating the Tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing*. Springer, 2017.

[65] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015.

[66] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 813–824. IEEE, 2013.

[67] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[68] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer. Tensorlab (Version 3.0). Available from http://www.tensorlab.net, March 2016.

[69] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003.

[70] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting Program Semantics to Place Data in Hybrid Memory. In *PACT*, 2015.

[71] K. Wu, Y. Huang, and D. Li. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[72] Kai Wu, Jie Ren, and Dong Li. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[73] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 331–345, New York, NY, USA, 2019. ACM.

[74] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*, 2019.

[75] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[76] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, 2012.

[77] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *International Conference on Supercomputing (ICS)*, 2017.