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

Jiawen Liu*
jliu265@ucmerced.edu
University of California, Merced

Jie Ren
jren6@ucmerced.edu
University of California, Merced

Roberto Gioiosa
roberto.gioiosa@pnnl.gov
Pacific Northwest National Laboratory

Dong Li
dli35@ucmerced.edu
University of California, Merced

Jiajia Li
jiajia.li@pnnl.gov
Pacific Northwest National Laboratory, William & Mary

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 sparse accumulator. 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.

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**.

Keywords: sparse tensor contraction, tensor product, multi-core CPU, non-volatile memory, heterogeneous memory

*This work was done when the author was an intern at PNNL.

PPoPP '21, 2/27 – 3/3, 2021, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441581

1 Introduction

Tensors, especially those high-dimensional sparse tensors are attracting increasing attention, because of their popularity in many applications. High-order sparse tensors have been studied well in tensor decomposition on various hardware platforms [9, 27, 35–38, 41, 50, 51, 54, 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 [4, 18, 31, 39, 58, 59], still lacks 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 the multiplication of a sparse tensor and 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 they are not efficient for SpTC. The first approach, using an extra symbolic phase [47] 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 exact same input is usually computed only once in a long sequence of tensor contractions [4]. 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

Corrected Version of Record. V.1.1. Published March 1, 2021.

very difficult because its more contract dimensions (see Section 2.2) make the prediction less accurate, using the existed prediction algorithms [2, 11].

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 are used to address irregular memory accesses in SpTC, are more expensive than that in SpGEMM. Our evaluation shows that they take 54% of SpTC execution time 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 [27, 35, 37, 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 performance drop. This memory capacity problem is becoming more serious in those HPC applications with increasing dimension size in tensors [4, 10, 15, 18, 45, 53, 66].

To address the first two challenges, we propose Sparta (Algorithm 2) with performance optimizations conducted 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 challenge of unknown output. 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 challenge of irregular memory accesses, 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 spatial 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

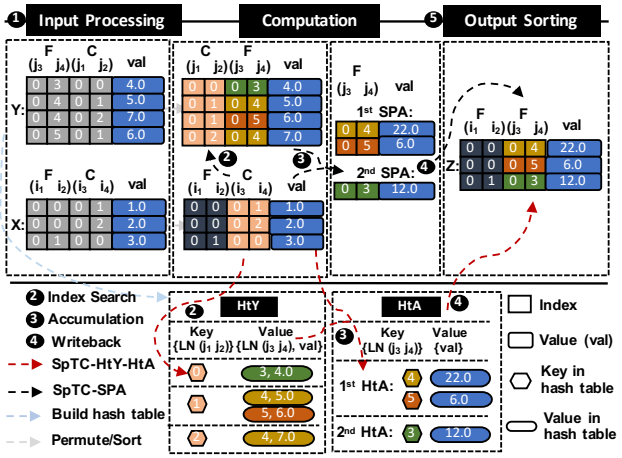

Figure 1. Workflow of the traditional SpTC-SPA and Sparta on $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$.

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 are placed in DRAM and the rest reside in PMM with large memory capacity of several TBs. 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 the 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 of 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 [55, 70, 80] or software-based page hotness tracking [1, 13, 24, 25, 72, 74, 76, 81]).

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 is 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).

[1] https://github.com/pnnl/HiParTI/tree/sparta

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 $\mathcal{X}$

- Evaluating with 15 datasets, Sparta brings 28 − 576× 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, in which most of its elements are zeros, is common in various applications. 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 [7] and TensorLab [68] (Refer to Section 3.2 for more reasons). A non-zero 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 non-zero value in a one-dimensional array *val*.

2.2 Sparse Tensor Contraction

Tensor contraction, a.k.a. tensor-times-tensor or mode-$(\{n\}, \{m\})$ product [10], 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 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 higher latency and lower 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 are 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

Algorithm 1: *SpTC-SPA*: Sparse tensor contraction of *Example 2:* $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$, extended from SpGEMM [20] 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 X, Y if not yet;
2  for X(i1, i2, :, :) in X do
3      Initiate a sparse accumulator SPA
4      for Non-zero x(i1, i2, i3, i4) in X(i1, i2, :, :) do
5          for Non-zero y(i3, i4, j3, j4) in Y(i3, i4, :, :) do
6              v = x(i1, i2, i3, i4) × y(i3, i4, j3, j4)
7              if SPA(j3, j4) exists then
8                  Accumulate SPA(j3, j4)+ = v
9              else
10                 Append v to SPA
11     Write SPA back to Z(i1, i2, :, :)
12 Permute and sort Z as needed
13 return Z
```

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 (but not permutation) 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 in the computation stages (see Sections 3.2 and 3.5 for details). Depending on the needs, sorting could be acted on $\mathcal{Z}$ after the computation stages, using the quick sort algorithm. This could avoid potential sorting when using $\mathcal{Z}$ as an input for any subsequent SpTC computations. In our algorithms and evaluation, sorting on $\mathcal{Z}$ is considered by default to get a thorough understanding of all stages.

3.2 Sparse Accumulator for High-order Sparse Tensors

Sparse accumulator (SPA) is a popular approach in sparse matrix-sparse matrix multiplication (SpGEMM) [19, 20], 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 $\mathcal{Y}$ 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. As shown in Algorithm 1, we loop all non-zeros of $\mathcal{X}$ by units of sub-tensors (see 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 of $O(nnz_Y)$ because of searching all non-zeros of $\mathcal{Y}$ in the worst case. To solve the multi-dimensional index search challenge, we construct $\mathcal{Y}$ as a hash table discussed 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 generalized, compressed sparse fiber (CSF) [65] format as follows. For example, we can directly 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 $\mathcal{Y}$ is not 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 accumulates 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 $SPA(0, 3)$ already exists, this product is added; Otherwise, the product along with its indices $(0, 3)$ are appended to *SPA*.

In Algorithm 1, since every $\mathcal{X}(i_1, i_2, :, :)$ independently accumulates to $\mathcal{Z}(i_1, i_2, :, :)$, *SPA* is allocated for each sub-tensor of $\mathcal{X}$. For each non-zero $x(i_1, i_2, i_3, i_4)$, if $\mathcal{Y}(i_3, i_4, :, :)$ is found by the index search, all non-zeros in $\mathcal{Y}(i_3, i_4, :, :)$ are stored contiguously and have good spatial data-locality due to the permutation and sorting of $\mathcal{Y}$ in the input processing stage. Since every non-zero in $\mathcal{Y}(i_3, i_4, :, :)$ computes with $x(i_1, i_2, i_3, i_4)$, $\mathcal{X}$ gets good temporary data-locality. If $SPA(j_3, j_4)$ already exists, the product v is added; 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 of $O(|SPA|)$ (|SPA| is 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 the index search stage, which is optimized with hash tables discussed in Section 3.4.

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

To solve the challenge of the unknown output size, traditionally two approaches, a two-phase method with symbolic and numeric phases [47] and a loose upper-bound size prediction [2, 11], have been introduced. The symbolic phase counts the number of non-zero elements of the output, which is expensive. Then, a precise memory space is allocated to perform the computation (the numeric phase). The approach

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

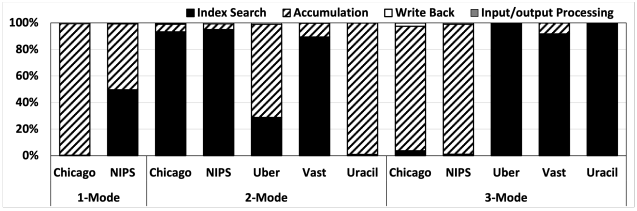

Figure 2. Percentage of execution time breakdown of *SpTC-SPA* (Algorithm 1).

of loose upper-bound size prediction allocates large enough memory based on probabilistic or upper bound prediction, which is more than sufficient for the output. In *SpTC-SPA*, we use dynamic vectors for the *SPA* and output tensor, like the progressive method [19] but more precise. The total time complexity of *SpTC-SPA* is

$$T_{SPA} = O(nnz_X log(nnz_X) + nnz_Y log(nnz_Y)) + O(2 \times nnz_X \times nnz_Y + nnz_Z) + O(nnz_Z log(nnz_Z)) \quad (3)$$

where the three terms correspond to the time complexity of input processing, *computation* with index search, accumulation, 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 the computation stages. Stages ❶ and ❺, shown together as *input/output processing*, take less than 1% of execution time of the algorithm. Compared to the two-phase method, our *SpTC-SPA* approach significantly reduces the input processing time; Compared to the prediction methods, *SpTC-SPA* can significantly 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 operation. Figure 2 for all of our test cases and Eq. 3 show that the stages ❷ and ❸ are the performance bottleneck. We focus on these two stages for performance optimization in Sections 3.2 to 3.5.

3.3 Hash Table-Represented Sparse Tensor

To address the problems of the multi-dimensional index search and inherit good data locality from *SpTC-SPA*, we propose to represent the input tensor $\mathcal{Y}$ with hash table.

Figure 1 depicts the process of converting $\mathcal{Y}$ represented in the 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 in 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 as the values in HtY. Since the keys in 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

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** $LN(C_{nz}^X)$ *is not found in* HtY **then**
9 continue
10 **for** $(LN(F_{nz}^Y), val_{nz}^Y)$ **in** $(LN(F^Y), V^Y)$ *of* HtY **do**
11 $v = val_{nz}^X * val_{nz}^Y$
12 **if** $LN(F_{nz}^Y)$ *is found in* HtA **then**
13 Accumulate $val_{nz}^{HT} += v$
14 **else**
15 Insert $(LN(F_{nz}^Y), v)$ to HtA
16 Form (F_{nz}^X, F_{nz}^Y) as coordinates and val_{nz}^{HT} 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** $\mathcal{Z}$

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 $3 = 0 \times J_4 + 3$. Having unique identifiers is extremely important for a fast hash table search. The large-number representation obtains unique numbers for every tuple of keys in HtY. As a result, the index search becomes faster on HtY by doing integer comparison for key comparison. To create HtY from $\mathcal{Y}$ in the COO format, we use the separate chaining hash table [71] with fix-sized buckets to distribute the keys. Compared to the COO format, the contract indices have no duplication due to the unique key feature of the hash table, which reduces the index search space. To maintain the good spatial data locality from Algorithm 1, we adopt dynamic arrays to store the non-zeros having the same key in $\mathcal{Y}$.

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 in Algorithm 2 are in the same order as those in Algorithm 1. The first and second loops enumerate sub-tensors in $\mathcal{X}$ and non-zeros in the sub-tensor using ptr_F to indicate locations. 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 in HtY respectively in Line 3.

For each non-zero element nz, we use $LN(C_{nz}^{X})$, the large-number representation of the contract indices C_X in $\mathcal{X}$ to search in HtY (Line 8). Compared with the linear search in *SpTC-SPA* with the complexity of $O(nnz_Y)$, the time complexity of hash table search on HtY is significantly reduced to $O(1)$ [71]. We also optimize input processing. The COO-to-hashtable conversion is faster than permutation and sorting of $\mathcal{Y}$ (i.e., $O(nnz_Y)$ versus $O(nnz_Y log(nnz_Y))$).

Our proposed hash table-represented sparse tensor with the large-number compressed keys significantly improves the SpTC performance by efficiently addressing the multi-dimensional index search issue and maintaining temporary and spatial 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

The hash table [3, 47–49], hashmap [12], and heap [6] are popular data structures to represent the accumulator in state-of-the-art SpGEMM research, where the hash table performs best according to prior evaluations [47]. As mentioned in Section 3.2 and Figure 2, the stage ❸ in *SpTC-SPA* could dominate the performance of an SpTC. To efficiently accumulate the intermediate results, we propose a hash table-based accumulator HtA, illustrated in Figure 1. We take the free indices of $\mathcal{Y}$, $(0, 3)$, as the key and refer to the intermediate result as the values in the hash table. The separate chaining hash table and the large-number representation LN are also adopted here for fast key matching and hash search.

We observe that the key in HtA ($(0, 3)$ in Figure 1) is the same as the free indices in $\mathcal{Y}$ in the value tuples of HtY. To avoid the key conversion for HtA, we convert the free indices of $\mathcal{Y}$ to the large-number representation in the stage ❶ (Line 3 in Algorithm 2). We directly retrieve the keys from the values of HtY, avoiding indices-key conversion between HtY and HtA during computation. As depicted in Figure 1 and Algorithm 2, the accumulation performs similar to *SpTC-SPA* but on the 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 the two hash tables HtY and HtA based on large-number representation respectively. Sparta solves the multi-dimensional index search challenge (Section 1), gets faster processing for input $\mathcal{Y}$, extracts unnecessary index computation/conversion out of the computation, while maintains the good data locality shown in *SpTC-SPA*, to reduce the SpTC execution time. The total time complexity of Sparta:

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

where nnz_{Favg} is the average size of all sub-tensors (e.g. $\mathcal{Y}(j_1, j_2, :, :)$ in Algorithm 1). The three terms in Eq. (4) correspond to the time complexity of the 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 are found in Section 5).

3.5 Parallelization

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

In the computation stages, we parallelize the outermost loop for sub-tensors 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. Because of the dynamic output structure, directly writing the intermediate, thread-local SPA or HtA to $\mathcal{Z}$ is not feasible. We introduce thread-local dynamic Z_{local} in Algorithm 2 to write the intermediate results. In particular, after a thread completes its execution, we have the size of Z_{local} which can be used to allocate the space for $\mathcal{Z}$. Then all threads write their 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 the performance of the stage ❹ with the cost of an 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 to major data objects in Sparta (Algorithm 2). We summarize memory access patterns (sequential/random and read/write) in Table 2. We consider six major data objects in the five stages (i.e., input processing, computation (combining index search, accumulation, writeback) and output sorting). The six major data objects are 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 the six data objects with the tensor Nell-2 (2-Mode contraction)

Table 2. Memory access patterns associated with data objects in five stages ("Ran" = Random; "Seq" = Sequential; "RW" = Read-Write; "RO" = Read-Only; "WO" = Write-Only).

Stages	Data Objects					
	$\mathcal{X}$	$\mathcal{Y}$	HtY	HtA	$\mathcal{Z}_{local}$	$\mathcal{Z}$
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

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 their impact on SpTC performance. Our baseline is the Sparta execution time when placing all data in DRAM, which achieves the best performance. We perform six tests: each one by placing only one data object in PMM, while leaving the others in DRAM. We have three interesting observations to guide our data placement in HM.

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 input processing stage 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 accumulation stage 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 input processing stage is sequential read-only, and placing it on PMM causes ignorable performance loss; In contrast, the memory access pattern associated with HtY in the index search stage 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 [73, 79]); 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 example, placing $\mathcal{X}$ and $\mathcal{Y}$ 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 (not seen in DRAM). In DRAM, both read and write, and sequential and random accesses have small performance difference. 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. $\mathcal{X}$ and $\mathcal{Y}$ are always on PMM, because of the observation 3. For the other four data objects,

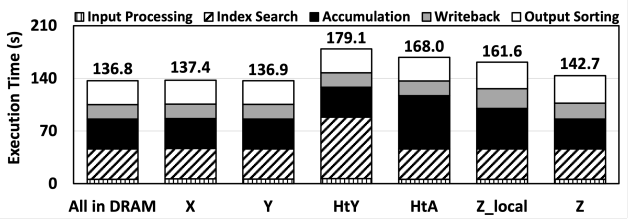

Figure 3. Performance after placing a data object in PMM while leaving others in DRAM. The x-axis shows the data object placed in PMM. "All in DRAM" means all data objects are placed in DRAM.

we decide their placement in DRAM, following the priority of $HtY \succ HtA \succ \mathcal{Z}_{local} \succ \mathcal{Z}$, according to their importance to the performance summarized from the characterization results. For each of the four data objects, we make the best efforts to place them into DRAM. This means that given a data object, if there is remaining DRAM space after excluding the memory consumed by the data objects with higher priority, 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 Eq. 5 based on tensor information and knowledge on data structures used in HtY. In Eq. 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; $nnz_{\mathcal{Y}}$ is the number of non-zero elements in $\mathcal{Y}$; N_Y is the number of modes of $\mathcal{Y}$.

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

Eq. 5 includes the memory consumption for metadata (i.e., the pointers pointing to each bucket in the hash table, modeled as $Size_{ep} \cdot \#Buckets_{HtY}$); Eq. 5 also includes the memory consumption for storing all non-zero elements of $\mathcal{Y}$ 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 Eq. 5, we must know $nnz_{\mathcal{Y}}$ and $\#Buckets_{HtY}$. $nnz_{\mathcal{Y}}$ as a tensor feature is typically known; $\#Buckets_{HtY}$ is defined by the user, and hence is known.

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

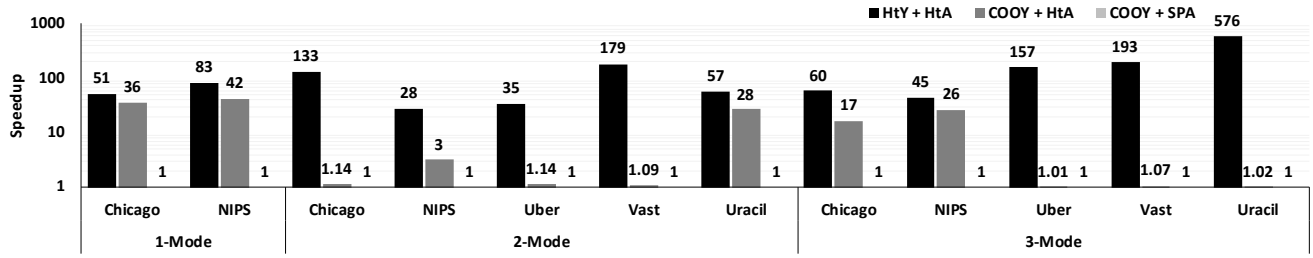

Figure 4. Speedups of HtY+HtA (i.e., Sparta) and COOY+HtA over COOY+SPA (i.e., SpTC-SPA) for SpTCs on Chicago, NIPS, Uber, Vast and Uracil with 1-mode, 2-mode and 3-mode.

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

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

Eq. 6 gives an upper bound, because we do not know the exact number of non-zero elements in $\mathcal{Y}$ that have the same contract indices as those in $\mathcal{X}$; 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 Eq. 6, we must know nnz^X_{Fmax} and nnz^Y_{Fmax}. nnz^X_{Fmax} and nnz^Y_{Fmax} are known after the input processing stage, and the dynamic allocation of *HtA* can happen after the input processing stage but before the index search stage where *HtA* is accessed. Hence, Eq. 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 $\mathcal{Z}_{local}$. The memory consumption of $\mathcal{Z}_{local}$ can be estimated after *HtA* is filled (Line 16 in Algorithm 2) and before memory allocation for $\mathcal{Z}_{local}$ happens. The memory consumption of $\mathcal{Z}_{local}$ is equal to the size of *HtA* plus the size of $F^X_{nz} \cdot nnz_{HtA}$, where F^X_{nz} 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 [1, 13, 24, 25, 56, 57, 72, 74, 76, 81] or manage DRAM as a hardware cache for PMM [42, 55, 70, 80] 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., the hardware-based Memory mode and software-based IAL [77]) 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 show the same priority for data placement (i.e., $HtY > HtA > \mathcal{Z}_{local} > \mathcal{Z}$). However, there are four cases showing different priorities (i.e., $HtA > HtY > \mathcal{Z}_{local}$ and $\mathcal{Z}$). For those uncommon cases, we can use the same method to determine data placement; The above methods to determine the sizes of the data objects are still valid.

Table 3. Characteristics of sparse tensors in the evaluation.

Tensors	Order	Dimensions	#Non-zeros	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}

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 at 2.6 GHz frequency on a 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 at 2.3 GHz frequency. The 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. Similar to recent work ([25], [72], [75]), we use a one-socket evaluation to highlight the data movement across DRAM and Optane. Each workload is run 10 times and we report the average execution time.

Datasets and expression. We use sparse tensors summarized in Table 3 and ordered by modes and non-zero density. Those tensors are derived from real-world applications. The tensors are included in FROSTT [62]. Tensor Uracil [4, 14] 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 expressions, we use a "*" to distinguish. For example, Chicago and Chicago* are the same tensors with different expressions.

5.2 Overall Performance

Figure 4 shows the performance of using HtY+HtA (i.e., Sparta), COOY+HtA and COOY+SPA (i.e., SpTC-SPA) on the 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 by $1.4 - 565\times$. The results show that HtY is more efficient than COOY. Also, we found that COOY+HtA significantly outperforms COOY+SPA by $1\% - 42\times$. The results demonstrate that HtA is more 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 the index search stage dominates the total execution time (99.3%) and the total execution time of this case is larger (1072 seconds) than that of others. Because of the time complexity difference between HtY and COOY in the index search stage, the larger execution time SpTC spends, the larger performance improvement Sparta can achieve. In Figure 2, the total execution time is dominated by the index search and accumulation stages in COOY-SPA (99.6%). Since the execution time of the index search and accumulation is significantly reduced by Sparta, the execution time of the index search and accumulation stages might not be the bottleneck of an SpTC. In our experiments with Sparta, the time in the index search stage accounts for 4.7%; the time of the accumulation stage is 61.6%; the time of the writeback stage is 9.6%; the input processing stage accounts for 3.3% and the output sorting stage is 20.8%.

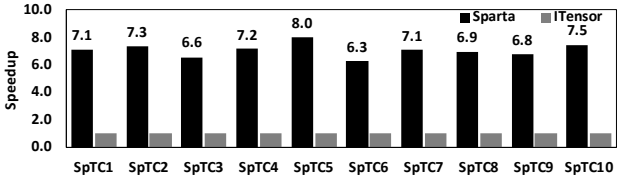

Figure 5. Speedups of Sparta over ITensor on Hubbard-2D model using different SpTC expressions with different sparse input tensors.

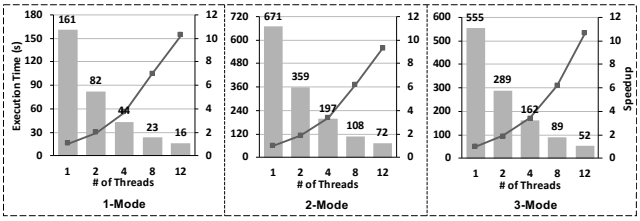

Figure 6. Thread scalability of parallel Sparta on SpTCs on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode.

5.3 Performance Comparison to ITensor

In this experiment, we compare the performance of Sparta and ITensor. ITensor [18] 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 [17]. SpTC expressions with different tensors (SpTC1 to SpTC10) are from a well-known quantum physics model (Hubbard-2D) [16] in ITensor [17], and those tensors are formed by cutting off [3] values smaller than 1×10^{-8}. More details of those tensors are shown in Table 4 in Appendix. We choose ITensor as a representative for comparison rather than others (such as libtensor [45], TiledArray [53], CTF [66] and TACO [30]), because libtensor only supports sequential block-wise SpTC [45], 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 also demonstrate that Sparta can 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

[3] Other truncating methods will be considered in the future.

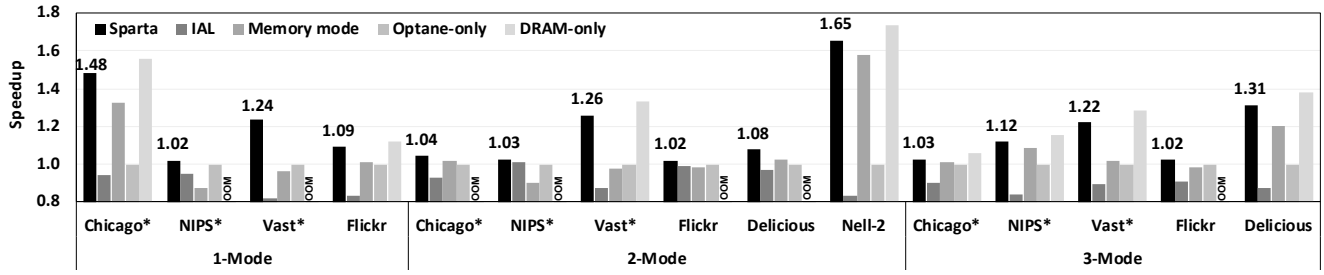

Figure 7. Speedups of Sparta, IAL, Memory mode and Dram-only over Optane-only for SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

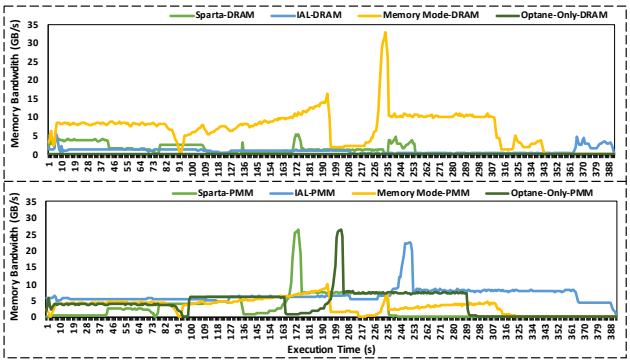

Figure 8. Memory bandwidth of Sparta, IAL, PMM Memory mode and Optane-only on Vast with 1-mode SpTC.

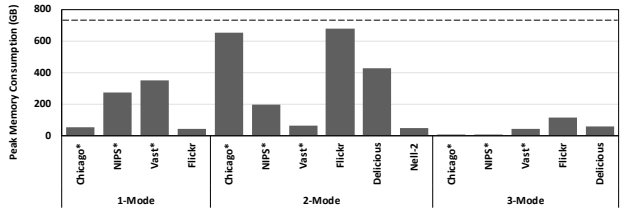

Figure 9. Peak memory consumption of SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

with 3-mode using 12 threads. Different stages have different thread scalability. Evaluation with 15 datasets using Sparta shows that the average speedup of parallel execution over sequential execution is: 10.4 × in the index search stage; 10.9 × in the accumulation stage; 9.5 × in the writeback stage; 6.8 × in the input processing stage and 6.2 × in the output sorting stage. The thread scalability in the stages of input processing and output sorting is not as good as that of the computation stages (i.e., index search, accumulation and writeback stages). However, the SpTC is always dominated by the computation stages. Thus, Sparta achieves high thread scalability overall.

5.5 Sparta on Heterogeneous Memory Systems

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) [77]), the hardware-managed cache approach (i,e, PMM Memory mode), Optane-only (i.e., the AppDirect mode 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 [78]. Figure 9 shows the peak memory consumption of SpTCs in the experiment.

As shown in Figure 7, Sparta outperforms IAL by 30.7% 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 respectively. Furthermore, Sparta is comparable to the DRAM-only approach with only 6% performance difference. 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, with the Chicago* with 3-mode, if we place all data objects to DRAM (i.e., DRAM-only), the performance improvement is only 6%, compared to Optane-only.

In Figure 8, we observe that the average PMM memory bandwidth of IAL is larger than that of Sparta. This is because IAL causes undesirable data movement and such data movement consumes higher PMM memory bandwidth. The average DRAM memory bandwidth of PMM memory mode is larger than that of Sparta, because PMM Memory mode manages DRAM as a 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 [5, 21, 23, 28, 29, 32, 33, 40, 46, 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, and then do multiplication by calling dense BLAS linear algebra and have the output tensor pre-allocated using domain knowledge or a symbolic phase [22, 26, 52, 53, 61], such as libtensor [15, 45], TiledArray [53], and Cyclops Tensor Framework [34]. 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% of non-zero density. Our work is also valuable for deep learning when the sparsity is introduced because of 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) [50, 51], flagged COO (F-COO) [41], and hierarchical coordinate (HiCOO) [37] for general sparse tensors, and mode-generic and -specific formats for structured sparse tensors [8]. We choose COO format in this work as a start because CSF format 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 the 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 [2, 3, 6, 12, 20, 30, 43, 47–49]. Our hash table implementations can be improved by using more advanced algorithms in [3, 44, 48, 49].

Data management on heterogeneous memory systems attracts a lot of attention recently. Many research efforts [1, 13, 24, 25, 72, 74, 76, 81] use a software-based solution to track data objects or page hotness to decide data placement on HM; Many research efforts [42, 55, 70, 80] 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 implement 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.

Acknowledgment

We thank Dr. Miles Stoudenmire and Dr. Matthew Fishman's help for using ITensor software and Dr. Ajay Panyala for helping us obtain the Uracil tensor and cutoff information. This research is partially funded by US National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194). This research is also partially funded by the US Department of Energy, Office for Advanced Scientific Computing (ASCR) under Award No. 66150: "CENATE: The Center for Advanced Technology Evaluation" and the Laboratory Directed Research and Development program at PNNL under contract No. ND8577. The Pacific Northwest National Laboratory (PNNL) is a multiprogram national laboratory operated for DOE by Battelle Memorial Institute under Contract DE-AC05-76RL01830.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. ITensor: A C++ library for efficient tensor network calculations. Available from https://github.com/ITensor/ITensor, August 2020.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. Model-driven sparse cp decomposition for higher-order tensors. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 1048–1057. IEEE, 2017.

[36] 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*, IA^3 '16, pages 26–33, Piscataway, NJ, USA, 2016. IEEE Press.

[37] 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.

[38] 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.

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

[40] 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.

[41] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. A unified optimization approach for sparse tensor operations on gpus. In *2017 IEEE international conference on cluster computing (CLUSTER)*, pages 47–57. IEEE, 2017.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *IEEE International Symposium on High Performance Computer Architecture*, 2021.

[57] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Neurips*, 2020.

[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] Yang Shi, Uma Naresh Niranjan, Animashree Anandkumar, and Cris 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. IEEE, 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] Wikipedia. Hash table. https://en.wikipedia.org/wiki/Hash_table, July 2020.

[72] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[73] Kai Wu, Jie Ren Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *USENIX Conference on File and Storage Technologies*, 2021.

[74] 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.

[75] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 31. IEEE Press, 2018.

[76] 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.

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

[78] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Repository of Nimble Page Management for Tiered Memory Systems in ASPLOS2019. Available from https://github.com/ysarch-lab/nimble_page_management_asplos_2019, July 2020.

[79] 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.

[80] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 337–344. IEEE, 2012.

[81] 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.

A Appendix

The sparse tensors are derived from Hubbard-2D in ITensor and summarized in Table 4 which includes the order, dimensions, #Non-zeros, density and #Blocks of both input tensors.

B Artifact Appendix

B.1 Build requirements

- GNU Compiler (GCC) (>=v7.5)
- CMake (>=v3.0)
- OpenBLAS
- NUMA

You may use the following steps to install the required libraries:

- OpenBLAS
  git clone https://github.com/xianyi/OpenBLAS
  cd OpenBLAS
  make -j
  mkdir path/to/OpenBLAS_install
  make install PREFIX=path/to/OpenBLAS_install
  Append export OpenBLAS_DIR=path/to/OpenBLAS_install to /.bashrc
- CMake
  sudo apt-get install cmake
- NUMA
  sudo apt-get install libnuma-dev
  sudo apt-get install numactl

B.2 Download and Set Up Projects

Download

git clone https://github.com/pnnl/HiParTI/tree/sparta (Sparta)
git clone https://gitlab.com/jiawenliu64/ial (IAL)
git clone https://gitlab.com/jiawenliu64/tensors (Datasets)

Build

cd sparta & ./build.sh

Set the Path Environments

You can execute the commands, e.g., export SPARTA_DIR= path/to/sparta, prior to the execution or append these commands to /.bashrc.

SPARTA_DIR (path/to/sparta, e.g., /home/ae/sparta)
IAL_SRC (path/to/IAL, e.g., /home/ae/ial/src)
TENSOR_DIR (path/to/tensors, /home/ae/tensors)

You can execute a command like "export EXPERIMENT_ MODES=x" to set up the environment variable for different test purposes. (This step has been included in our scripts below, so you don't need to explicitly specify it.)

export EXPERIMENT_MODES=0: COOY + SPA
export EXPERIMENT_MODES=1: COOY + HtA
export EXPERIMENT_MODES=3: HtY + HtA
export EXPERIMENT_MODES=4: HtY + HtA on Optane

A Test Run

On a general multicore CPU server with Linux:
./sparta/run/test_run.sh

On a server with Intel Optane DC PMM:
./sparta/run_optane/test_run.sh

B.3 More Support

Tensor Contraction Parameters

You can check the parameters options with path/to/sparta/ build/ttt –help

Options:
-X FIRST INPUT TENSOR
-Y FIRST INPUT TENSOR
-Z OUTPUT TENSOR (Optinal)
-m NUMBER OF CONTRACT MODES
-x CONTRACT MODES FOR TENSOR X (0-based)
-y CONTRACT MODES FOR TENSOR Y (0-based)
-t NTHREADS, –nt=NT (Optinal)
–help

B.4 ITensor Results Generation

We have generated the sparse tensors and performance from ITensor library and stored them in itensor/results. If you want to recollect all these tensors, you can use the following steps.

- git clone https://gitlab.com/jiawenliu64/itensor (forked from ITensor repo, also provided in the "Artifact download URL" in the PPoPP AE submission.)
- export ITENSOR_DIR=path/to/itensor
- mkdir path/to/itensor_results & export ITENSOR_RES ULTS=path/to/itensor_results
- cd $ITENSOR_DIR & run.sh
- cd $ITENSOR_DIR/hubbard & OMP_NUM_THREADS =12 ./main 'parity' 1. After the execution, all results are stored in $ITENSOR_RESULTS. The result (execution time) is included in the second line of each generated file (e.g., tensor_2137.txt).

If you also want to convert the data to the .bin format as they are shown in path/to/itensor/results, you can use the following steps to process data using SPLATT, another sparse tensor library.

- git clone https://github.com/ShadenSmith/splatt
- ./configure –prefix=SPLATT_DIR & make & make install
- Replace all Block in tensor A to A-Block. For example, in vim, you can execute x,ys/Block/A-Block/g to replace from line x to line y.
- Replace all Block in tensor B to B-Block. For example, in vim, you can execute x,ys/Block/B-Block/g to replace from line x to line y.
- python path/to/sparta/output_scripts/gen_tns_itensor.py path/to/itensor_results/tensor_x.txt 0.00000001 for data tensor_x.
- path/to/splatt/build/Linux-x86_64/bin/splatt convert -t bin path/to/itensor_results/tensor_x_A.tns path/to/it ensor_results/tensor_x_A.bin for A in x.

Table 4. Characteristics of tensors of ITensor in the evaluation.

SpTC	Tensors	Order	Dimensions	#Non-zeros	Density	#Blocks	Tensors	Order	Dimensions	#Non-zeros	Density	#Blocks
1	X	5	$129 \times 4 \times 184 \times 24 \times 4$	109287	4.8×10^{-3}	10453	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
2	X	5	$129 \times 4 \times 184 \times 24 \times 4$	114877	5.0×10^{-3}	12044	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
3	X	5	$4 \times 129 \times 184 \times 24 \times 4$	114877	5.0×10^{-3}	12044	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
4	X	5	$4 \times 131 \times 4 \times 24 \times 413$	262218	6.3×10^{-3}	12345	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
5	X	5	$131 \times 4 \times 413 \times 36 \times 4$	377629	4.8×10^{-3}	17594	Y	4	$36 \times 24 \times 4 \times 4$	360	5.9×10^{-3}	218
6	X	5	$4 \times 131 \times 4 \times 24 \times 413$	268813	6.4×10^{-3}	13288	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
7	X	5	$131 \times 4 \times 413 \times 36 \times 4$	388132	5.2×10^{-3}	19367	Y	4	$36 \times 24 \times 4 \times 4$	360	5.9×10^{-3}	218
8	X	5	$4 \times 4 \times 131 \times 24 \times 413$	268813	6.5×10^{-3}	13288	Y	4	$24 \times 36 \times 4 \times 4$	360	6.9×10^{-3}	218
9	X	5	$4 \times 131 \times 413 \times 36 \times 4$	388132	5.2×10^{-3}	19367	Y	4	$36 \times 24 \times 4 \times 4$	360	5.9×10^{-3}	218
10	X	5	$4 \times 110 \times 4 \times 36 \times 486$	396193	6.4×10^{-3}	17152	Y	4	$36 \times 24 \times 4 \times 4$	360	5.9×10^{-3}	218

- path/to/splatt/build/Linux-x86_64/bin/splatt convert -t bin path/to/itensor_results/tensor_x_B.tns path/to/itensor_results/tensor_x_B.bin for B in x.