Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory

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

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

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

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

ABSTRACT

Sparse tensor contraction sequence has been widely employed in many fields, such as chemistry and physics. However, how to efficiently implement the sequence faces multiple challenges, such as redundant computations and memory operations, massive memory consumption, and inefficient utilization of hardware. To address the above challenges, we introduce Athena, a high-performance framework for SpTC sequences. Athena introduces new data structures, leverages emerging Optane-based heterogeneous memory (HM) architecture, and adopts stage parallelism. In particular, Athena introduces shared hash table-represented sparse accumulator to eliminate unnecessary input processing and data migration; Athena uses a novel data-semantic guided dynamic migration solution to make the best use of the Optane-based HM for high performance; Athena also co-runs execution phases with different characteristics to enable high hardware utilization. Evaluating with 12 datasets, we show that Athena brings 327-7362× speedup over the state-of-the-art SpTC algorithm. With the dynamic data placement guided by data semantics, Athena brings performance improvement on Optane-based HM over a state-of-the-art software-based data management solution, a hardware-based data management solution, and PMM-only by 1.58×, 1.82×, and 2.34× respectively. Athena also showcases its effectiveness in quantum chemistry and physics scenarios.

CCS CONCEPTS

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

KEYWORDS

Sparse tensor contraction sequences, sparse tensor product, multi-core CPU, non-volatile memory, heterogeneous memory

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

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460355

ACM Reference Format:
Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. 2021. Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3447818.3460355

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 [7, 26, 37–40, 43, 51, 52, 55, 69–71] with a focus on the product of a sparse tensor and a dense matrix or vector. The two sparse tensor contraction (SpTC) has been studied well [14, 20, 25, 36, 44, 49, 53, 54, 54, 67] where block-wise sparsity is the main focus. As the needs of element-/pair-wise sparsity emerge in applications from chemistry, physics and deep learning [4, 17, 31, 41, 62, 63], the recent work [44] studied element-wise SpTC. In essence, SpTC, a high-order extension of sparse matrix-matrix multiplication (SpGEMM), multiplies two sparse tensors along with their common dimensions.

Nevertheless, SpTC commonly is shown as sequences in quantum chemistry, quantum physics and deep learning [4, 17, 31, 41, 62, 63] as a foundation of coupled cluster single double (Triple), CCSD(T) [10], high-order tensor decomposition methods, etc. An SpTC sequence (SpTCSeq) performs a sequence of sparse tensor contractions which could be independent or have different dependency types (will be explained in Table 2). While sequences of tensor contraction have been studied [34] with a focus on independent contractions and limited dependency types, such as identical input and shared outputs , an SpTCSeq is still lack of sufficient research for element-wise contractions and other dependency types. For example, Type 1 dependency, the output tensor of an SpTC taken as an input in another SpTC, occurs in 85% contractions in CCSD(T) from the NWChem library and has not been studied yet. However, multiple challenges impede obtaining high performance for a whole SpTC sequence.

First, redundant computation and memory traffic in an SpTCSeq lead to performance issues since it shares data objects across different SpTCs. As shown in Table 2, an SpTCSeq could include four dependency types and some data objects are shared across different SpTCs (more details in Section 2.2). Computation and memory traffic on the shared data objects are performed repeatedly. For example, in the type that two SpTCs share an identical input tensor,

the processing on this input in the second SpTC can be avoided. Because of the shared data objects, computation and memory access on intermediate data objects are also performed repeatedly. For example, in the type that the output tensor of the first SpTC becomes the input tensor in the second SpTC, the intermediate results in the accumulation of the first SpTC can be directly reused to do the computation of the second SpTC, skipping multiple stages in the sequential execution. (Details in Section 3.2) This performance issue becomes severer when the redundant computation and memory access dominate the execution. Moreover, the performance suffers when we perform an SpTCSeq with a larger number of contractions.

Second, large memory consumption from large input and output tensors and intermediate results causes a memory capacity issue and 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 when generated with more non-zero elements than any of the input. The intermediate results could be large as well, especially in multi-threading environment where each thread has its local intermediate results. Compared to the well-studied sparse tensor times dense matrices/vectors [26, 37, 39, 70], 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. The memory capacity problem in an SpTCSeq becomes much more serious than in an individual SpTC. This memory capacity problem is especially pronounced in those HPC applications with increasing dimension sizes of tensors [4, 9, 14, 17, 49, 54, 72]. Expanding DRAM capacity is not cost-effective, while adding cheap but much slower Solid-State Drive (SSD) causes significant performance drop.

Third, an SpTCSeq suffers from inefficient hardware utilization due to the diverse computation and memory patterns of different stages in an SpTC. For example, the accumulation stage in an SpTC with tensor Disilane (in Section 5.1), the average memory bandwidth is only 19.3% of the peak memory bandwidth, while the average CPU utilization is 71.9%; for index search stage (proposed in [44]) in an SpTC with the same tensor Disilane, the average CPU utilization is only 34.2% while the memory bandwidth is 44.1%. Given a 2-SpTC sequence, if we simply run the SpTCs sequentially, the hardware (e.g., computing units or memory bandwidth) is not fully utilized; If we co-run stages in the same intensive pattern, e.g., both memory-intensive, the SpTCSeq suffers from resource (e.g., memory bandwidth) contention; How to efficiently arrange stages across SpTCs in a sequence to achieve efficient hardware utilization without resource contention is challenging.

To address the above challenges, we propose Athena, a high-performance framework for SpTC sequences. To address the first challenge, we introduce shared hash table-represented sparse accumulator. In particular, given two SpTCs, we adopt hash table-represented sparse accumulator with reusing intermediate results in the first SpTC and then perform index search in the second SpTC to eliminate finishing up stages of the first SpTC and the starting expense of the second SpTC. Moreover, we retain shared data objects across SpTCs to eliminate unnecessary input processing and data migration. We also introduce a hash table-represented sparse tensor summation to significantly increase the performance of summation stages which are widely used in SpTC sequences. Athena effectively avoids redundant computation and memory operations in an SpTCSeq with shared data objects.

To address the second challenge, we explore the persistent memory-based heterogeneous memory (HM). In particular, the emerging Intel Optane DC Persistent Memory Module (PMM) provides up to 9TB memory capacity per node, which can be leveraged to address the memory capacity problem faced by SpTCSeq. PMM has slightly inferior bandwidth and latency (compared to DRAM) but with much lower price. As a result, PMM is often paired with a small DRAM, such that frequently accessed data objects can be placed into DRAM while the rest reside in PMM with large memory capacity. PMM and DRAM builds a heterogeneous memory system.

The PMM-based HM raises a question on how to perform an SpTCSeq given limited DRAM space for high performance. Effectively placing data objects of an SpTCSeq in DRAM and PMM for high performance is critical to use PMM to address the memory capacity problem faced by SpTCSeq. To decide data placement on HM, the traditional solutions track page (or data) access frequency [2, 12, 22, 24, 58, 61, 78, 79, 82, 87] or manage DRAM as a hardware cache for PMM [46, 57, 76, 86], and then reactively place frequently accessed data objects into DRAM subject to the DRAM capacity constraint. However, those solutions are application-agnostic, and cause unnecessary and frequent data movement because of short-term variance in memory access patterns. The static data placement strategy [44] places data objects in DRAM or PMM without triggering dynamic migration in the middle of application execution. However, this strategy lacks the flexibility of handling irregular memory access patterns but with certain temporal locality.

Athena addresses the above problem by introducing a data-semantics guided data placement. This solution strikes a good balance between the static and dynamic data placement. In particular, it leverages data semantics to guide dynamic data placement. Instead of tracking the number of memory accesses at runtime as in the traditional dynamic data placement, we use the algorithm knowledge to reason the numbers of memory accesses (or hotness) at data object level during the construction of critical data structures in SpTCSeq, and then associate those numbers with data objects. After using the data semantics to identify those data objects, Athena is able to use hotness information to guide dynamic data placement.

To address the third challenge, we introduce stage parallelism for an SpTCSeq. We first characterize computation and memory behaviors of different stages in an SpTCSeq. Next, we co-run those stages in an SpTCSeq with respect to their integer operations (IOP)-, floating point operations (FLOP)-, or memory-intensive patterns, to avoid resource contentions and meanwhile improve the utilization of CPU and memory bandwidth. Hyperthreading technique is used for data prefetching and higher memory bandwidth usage to gain better overlapping between two stages. For exascale problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to its capability to solve large sparse tensors on each single node.

Our main contributions are summarized as follows:

- We introduce the first, high-performance SpTCSeq system for element-wise sparse tensor contraction sequence, named Athena. (Section 3).

Table 1: List of symbols and notation.

Symbols	Description
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$	Sparse tensors
$\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}$	Tensor contraction between two tensors
I, 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	#Non-zeros of sub-tensors of $\mathcal{X}$
ptr_F	Pointers for mode-F^X sub-tensor locations of $\mathcal{X}$
M^X	A set of all modes in $\mathcal{X}$
C^X	A set of contract modes in $\mathcal{X}$, $\{n\}$ in $\times_{\{n\}}^{\{m\}}$ contraction
F^X	A set of free modes in $\mathcal{X}$, $\lvert F^X \rvert + \lvert C^X \rvert = N_X$
m_{nz}^X	All mode indices of a non-zero element in $\mathcal{X}$
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 $\mathcal{X}$
V^X	A set of non-zero values in $\mathcal{X}$
v^X	Value of a non-zero element in $\mathcal{X}$

- We explore the emerging PMM-based HM to address memory capacity limitation suffered in the tensor computations, and use algorithm knowledge and data semantics to guide dynamic data placement (Section 4).
- Evaluating with 12 datasets, we show that Athena brings 327-7362 × speedup over the state-of-the-art SpTC algorithm. With the dynamic data placement guided by data semantics, Athena brings 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 1.58× (up to 2.09×), 1.82× (up to 2.58×) and 2.34× (up to 2.94×) respectively (Section 5).

2 BACKGROUND

A tensor can be treated 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., $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$ (a tensor with four modes), and $x_{i_1 i_2 i_3 i_4}$ is its (i_1, i_2, i_3, i_4)-element. Table 1 summarizes notation and symbols used in this work.

The element-wise sparse data is commonly found in various applications in data analytics, signal processing, recommendation systems, and deep learning [3, 4, 9, 23, 30, 31, 41, 56, 62, 63, 66]. Most of elements in sparse data are zeros. In order to save storage space, compact representations of the sparse tensor are proposed [39, 43, 52, 71]. We adopt the popular format, the coordinate (COO), in this work, which is widely used in popular tensor libraries, such as Tensor Toolbox [6], TensorLab [74], and TACO [29]. A non-zero element is stored as a tuple for its indices, e.g., (i_1, i_2, i_3, i_4) for a fourth-order tensor, in a two-level pointer array *inds*, along with its non-zero value in a one-dimensional array *val*.

2.1 Sparse Tensor Contraction

Tensor contraction. Tensor contraction (i.e., tensor-times-tensor or mode-$(\{n\}, \{m\})$ product [9]) is an extension of matrix multiplication. It is represented as $\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}$, where $\{n\}$ and $\{m\}$ are tensor modes to perform this product.

Table 2: Expression dependency between two SpTCs.

Type	Feature	Expressions
1	Output as input	$\mathcal{Z}' = \mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{Z}'$
2	Identical input & Different output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}'$ += $\mathcal{W} \times \mathcal{Y}$
3	Different input & Shared output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{V}$
4	Identical input & Shared output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{Y}$
5	Independent	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}'$ += $\mathcal{W} \times \mathcal{V}$

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 (represented as $C_X = \{3, 4\}$ and $C_Y = \{1, 2\}$), and the other modes are free modes. The operation in this example is represented as: $z_{i_1 i_2 j_3 j_4} = \sum_{i_3(j_1)=1}^{I_3(J_1)} \sum_{i_4(j_2)=1}^{I_4(J_2)} x_{i_1 i_2 i_3 i_4} x_{j_1 j_2 j_3 j_4}$.

Element-wise Sparse Tensor Contraction. Element-wise sparse tensor contractions (SpTC) emerges in the applications of chemistry and physics [4, 17, 31, 41, 62, 63], where both input and output tensors have element-wise sparsity. Sparta [44] is the state-of-the-art algorithm for an arbitrary-order, element-wise SpTC. Sparta introduces a hash table-based representation for input sparse tensors and a sparse accumulator for a single element-wise sparse tensor contraction. The Sparta SpTC algorithm contains five stages: input processing, index search, accumulation, writeback, output sorting stages. Refer to [44] for more details.

2.2 Sparse Tensor Contraction Sequences

Sparse tensor contractions sequence (SpTCSeq) is widely used in many methods. For example, SpTCSeq can be derived from the well-known Coupled Cluster Single Double (Triple), CCSD(T) [10], in chemistry [4] and from the notable Hubbard model [15] in physics [17]. Within an SpTCSeq, multiple SpTCs could have dependence between each other or be independent, and they might share some identical tensors in different ways.

We summarize common expression types of SpTCSeq in Table 2. Five types could exist for two arbitrary SpTCs. In Type 1, the output tensor of the first SpTC, $\mathcal{Z}'$, used as an input tensor of the second SpTC; In Type 2, both SpTCs have an identical input tensor $\mathcal{Y}$; In Type 3, the two SpTCs use different input tensors but generate the same output tensor $\mathcal{Z}$; In Type 4, both SpTCs use an identical input tensor $\mathcal{Y}$ and share the output tensor $\mathcal{Z}$; In Type 5, the two SpTCs are totally independent from each other. We quantify the occurrence percentage of the five types of SpTCSeq in CCSD(T) [10] from chemistry. Types 1-5 account for 85%, 9%, 6%, 8% and 91% of all SpTCSeq, respectively. Note that the sum of all types is more than 100%, because an SpTC equation could fall into more than one type. Besides, sparse tensor summation, the "+" operator in most expressions of Table 2, is common in sparse tensor contractions sequences (e.g., accounts for 90% in CCSD(T) [10] in chemistry). We will introduce our design of sparse tensor summation in Section 3.1, Types 1-4 SpTCSeq dependency in Section 3.2, and Type 5 dependency in Section 3.3.

2.3 Intel Optane DC Persistent Memory Module

The recent release of the Intel Optane DC Persistent Memory Module (PMM) is the first byte-addressed non-volatile memory (NVM)

Algorithm 1: *Athena*: sparse tensor contraction sequence for arbitrary-order data in the expression $\mathcal{Z} = \mathcal{X} \times_{C_X}^{C_Y} \mathcal{Y} \times_{F_Y}^{C_W} \mathcal{W} + \mathcal{Z}_{pre}$.

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_{N_X}}$, $\mathcal{Y} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_Y}}$, and $\mathcal{W} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_W}}$, contract modes C_X, C_Y, C_W, and output tensor $\mathcal{Z}_{pre}$ produced from previous SpTCs
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 and $\mathcal{W}$ to HtW
4 **for** f **in** $1, \ldots, N_F$ **do**
5 Initiate thread-local HtA and Shared-HtA
6 **for** nz **in** $ptr_F[f], \ldots, ptr_F[f+1]$ **do**
7 $Index\_Search(c_{nz}^X, HtY)$
8 $Accumulation(f_{nz}^Y, v^X, v^Y, v^{HtA})$
9 **for** (key, v^{HtA}) **in** HtA **do**
10 **if** key *is not found in* HtW **then**
11 continue
12 **for** $(LN(f_{nz}^W), v^W)$ **in** $(LN(F^W), V^W)$ *of* HtW **do**
13 **if** $LN(f_{nz}^W)$ *is found in* HtW **then**
14 Accumulate v^{SHtA} += $v^{HtA} * v^W$
15 **else**
16 Insert $(LN(f_{nz}^W), v^{HtA} * v^W)$ to Shared-HtA
17 Form (f_{nz}^X, f_{nz}^W) as coordinates and v^{SHtA} as non-zero value and append to $\mathcal{Z}_{local}$
18 Gather thread-local $\mathcal{Z}_{local}$ independently to $\mathcal{Z}$
19 Convert $\mathcal{Z}$ to HtZ with M^Z as keys,
20 **for** nz **in** $\mathcal{Z}_{pre}$ **do**
21 **if** $LN(m_{nz}^{Z_{pre}})$ *is not found in* HtZ **then**
22 Append $(LN(m_{nz}^{Z_{pre}}), v^{Z_{pre}})$ to HtZ
23 **else**
24 Append $(LN(m_{nz}^{Z_{pre}}), v^{Z_{pre}} + v^Z)$ to HtZ
25 Convert HtZ to $\mathcal{Z}$, and permute/sort $\mathcal{Z}$ if needed
26 **return** $\mathcal{Z}$

in the market. PMM can be configured to work in either *Memory* or *AppDirect* mode. In the Memory mode, DRAM is a hardware-managed, directly-mapped write-back cache to PMM and is transparent to applications. In the AppDirect mode, the placement of data objects on PMM and DRAM can be explicitly controlled by the programmers.

PMM can provide up to 6TB memory capacity on a single machine, but has 2.2-3.5× higher access latency and 2.7-6.2 × lower bandwidth than the traditional DRAM. Sparta [44] statically allocates data objects to either DRAM or PMM according to their memory access patterns. However, this simple static data placement does not work best for all data objects, especially data objects with random read/write memory access. Our work leverages the AppDirect mode with dynamic data management and leads to better performance than the Memory mode.

3 ALGORITHM DESIGN

This section introduces our SpTCSeq algorithm for the five dependency types in Table 2 and efficient sparse tensor summation.

3.1 Hash Table-Represented Sparse Tensor Summation

A general process of two element-wise sparse tensor summation is as follows. Given a sparse output tensor $\mathcal{Z}_{pre}$ produced from previous SpTCs or other operations (hidden in the "+=" operator in Table 2) and a sparse output tensor $\mathcal{Z}$ in the current SpTC, the sparse tensor summation performs three steps. First, a non-zero element along with its indices from $\mathcal{Z}_{pre}$ is selected. Next, the summation searches the corresponding non-zero element(s) in $\mathcal{Z}$ with the exact same tuple of indices. Finally, if the particular non-zero element in $\mathcal{Z}$ with the same indices is found, $\mathcal{Z}$ is updated with the sum of the two non-zero values under the tuple of indices. Otherwise, the non-zero element in $\mathcal{Z}_{pre}$ is appended to $\mathcal{Z}$ as a new element. Meanwhile, the non-zero elements of $\mathcal{Z}$ which are not updated during the summation remain the same. This process is expensive in the searching step due to multi-dimensionality of the tuple of indices as keys and dynamically updating $\mathcal{Z}$ especially with appending new non-zero elements from $\mathcal{Z}_{pre}$.

To address the above problems, we propose the hash table-represented sparse tensor summation for an SpTC. Figure 1 depicts our proposed approach as Summation, stage 5 (The rest stages will be explained in Section 3.2). It is extremely time-consuming to perform key matching on multi-dimensional tuples, especially when the tensor order is large high [44]. We adopt the hash table representation from the work [44] by first converting the sparse output tensor $\mathcal{Z}$ in COO format to a hash table-represented $Ht\mathcal{Z}$. The index tuples of $\mathcal{Z}$ are taken as the keys of HtZ naturally, different from the key construction in Sparta [44]. A large-number representation, noted as the LN function in Figure 1, is also leveraged to convert a sparse index tuple to a large and unique index. The index search is improved by 1) reduced searching space of the unique index keys of the hash table; 2) pinpointing the targeted index much faster than the traditional linear search approach with a constant algorithm complexity. To fast update $\mathcal{Z}$ and maintain good spacial data locality, we adopt dynamic arrays to construct the values of HtZ for the non-zeros having the same key. $Ht\mathcal{Z}$ is then converted back to $\mathcal{Z}$ as the final output. As shown in line 19 to 24 in Algorithm 1, Athena converts $\mathcal{Z}$ to HtZ and then iterates all non-zeros in $\mathcal{Z}_{pre}$. If the nnz index is not found in HtZ, Athena appends the key-value pair to HtZ. Otherwise, Athena appends the index along with the summed values to HtZ.

Our hash table-represented sparse tensor summation extends to support the fused multiplication and summation and plays a critical role for performance (in Section 5.3) when the size of output tensor is similar to or even larger than input tensors in an SpTC.

3.2 Shared Hash Table-Represented Sparse Accumulator

We observe that the traditional approach for Type 1 dependency (Output as input) in Table 2 leads to repeated and inefficient computations and data movement because the two SpTCs share some intermediate data objects. To address this problem, we introduce a shared hash table-represented sparse accumulator (named *Shared-HtA*). Figure 1 depicts the workflow of our Shared-HtA design. The first SpTC, $\mathcal{Z}' = \mathcal{X} \times \mathcal{Y}$, follows the five-stage computation

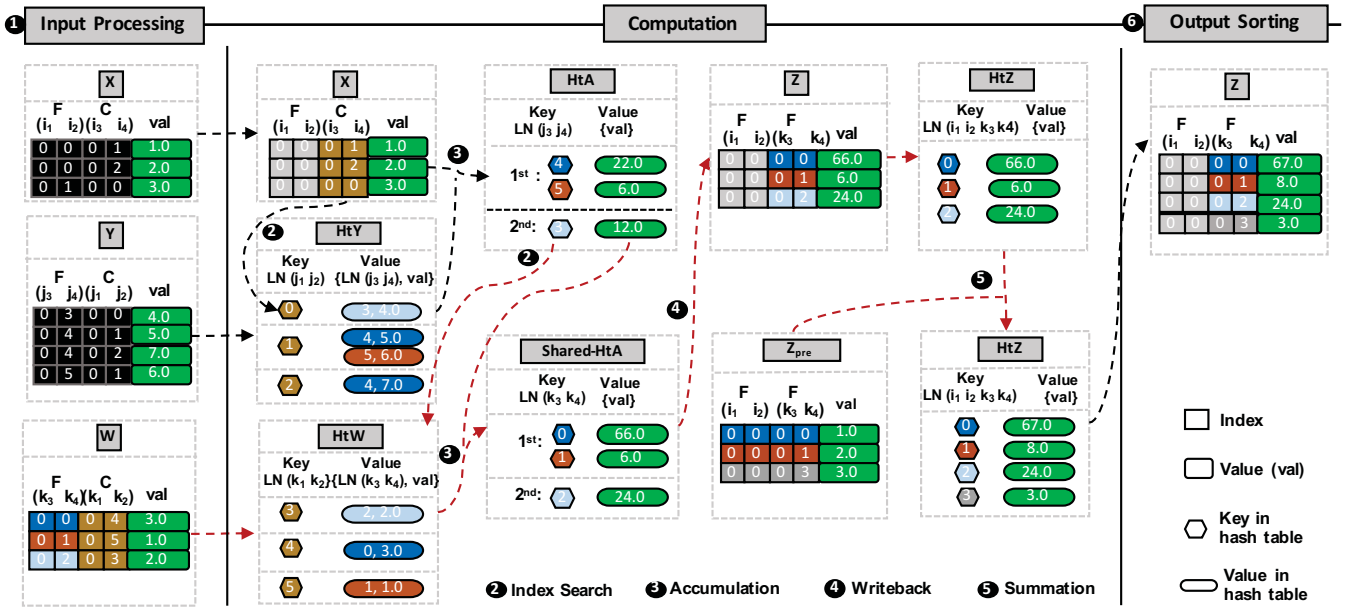

Figure 1: Workflow of Type 1 dependency of two SpTCs in Table 2, using shared hash table-represented sparse accumulator and hash table-represented sparse tensor summation (indicated by red arrows)

proposed in the work [44]: input processing, index search, accumulation, writeback, output sorting, stages 1-4 and 6. The hash table-represented summation is the new stage 5.

Once the index search and accumulation stages of the first SpTC are completed, we treat the free modes F_Y of $\mathcal{Y}$ in *HtA* to be the contract modes of the second SpTC. We then employ F_Y as the key to search the corresponding contract modes C_W in *HtW* in another index search stage for the second SpTC, $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{Z}'$. For example, in Figure 1, 3 is used as the key in the 2^{nd} *HtA* to search the corresponding contract modes (3) in *HtW*. Next, we generate the Shared-*HtA* to store the intermediate results during the accumulation stage of the second SpTC (2 and 24.0). Once the index search and accumulation of $\mathcal{X}$ with the same free modes are completed (i.e., index search and accumulation stages in both black and red arrows in Figure 1) , the intermediate results of Shared-*HtA* are converted back to COO format ((0, 2) tuple) and appended to $\mathcal{Z}$ along with its accumulated result (24.0) for summation and output sorting stages. Finally, the intermediate Shared-*HtA* is released to save memory space. This approach also leverages the same large-number representation approach in the work [44] and converts the input tensor $\mathcal{W}$ of the second SpTC to the hash table representation *HtW*. The full algorithm of Shared-*HtA* is illustrated in line 4 - 18 in Algorithm 1. Athena first completes the index search and accumulation in the first SpTC in line (5-8). Athena then iterates each key-value pair in *HtA* and leverages Shared-*HtA* to calculate and store the intermediate results (line 9-17). Finally, Athena gather thread-local $\mathcal{Z}_{local}$ independently to $\mathcal{Z}$ (line 18).

By employing the Shared-*HtA*, we eliminate multiple time consuming stages: appending intermediate results in the accumulation stage, writeback and output sorting stages of the first SpTC and input permutation/sorting in the input processing stage of the first SpTC's output and large-number conversion in the index search stage of the second SpTC. Therefore, our proposed Shared-*HtA* avoids the repeated computation and eliminate unnecessary data movement and hence significantly improves the performance and memory efficiency for an SpTCSeq in Type 1.

Types 2-4 dependency in Table 2 is less frequently occurred compared to Type 1. Type 4 has been studied in previous research [48], which converted to $\mathcal{Z}$ += $(\mathcal{X} + \mathcal{W}) \times \mathcal{Y}$ to improve performance through contraction fusion and replacing one tensor product with a summation operation. For an SpTCSeq in Types 2 and 3 dependency, we could utilize a simple strategy to avoid redundant data movement between DRAM and storage (or PMM) if the DRAM space is adequate. For Type 2, after completing the first SpTC, the identical input tensor $\mathcal{Y}$ remains in DRAM; similarly for the shared output tensor $\mathcal{Z}$ for Type 3.

3.3 Stage Parallelism

We propose stage parallelism to better utilize machine resources, such as CPU and memory bandwidth. This is different from the prior research [34, 48] which uses strategies, like compiler-based tensor contraction expression generator [48] or hand-tuned optimization [34], to obtain independent expressions of an SpTCSeq in an optimal order following dependency types. After resolving different types of dependent SpTCSeq, we treat an SpTCSeq in Types 1-5 as a single simple/compound, independent SpTC in this section.

Stage characterization. We first explore the characteristics of the six stages (i.e., input processing, index search, accumulation, writeback, summation and output sorting) in the SpTC algorithm (Algorithm 1) and categorize them into IOP-, FLOP-, and memory-intensive behaviors. We calculate the compulsory number of integer operations (IOPs), floating point operations (FLOPs), and memory

traffic of the stages. The word "compulsory" means the minimum requirements of operations or memory traffic assuming an infinite cache size, which gives a fundamental idea of an algorithm behavior and has been used in performance model analysis [77]. Diverse IOPs, FLOPs, and memory traffic behaviors have been observed in the six stages.

In particular, we observe that three stages, namely input processing, index search and output sorting, dominated by integer operations (IOPs). Sorting and/or permutation, the primary components of input processing and output sorting stages, have frequent index comparison and exchanging. Index search performs search on index tuples, thus only IOPs are needed. These stages are referred to as *IOP-intensive stages*. Accumulation and summation stages, referred to as *FLOP-intensive stages*, consist of the core floating point operations of a tensor contraction. The writeback stage has pure memory access, named *memory-intensive stages*. IOP-, FLOPS-, and memory-intensive stages utilize different computing units or memory components to fulfill, which makes it possible to parallelize them from multiple SpTCs to improve hardware utilization. The above observations on stage characterization drive our design.

Concurrency control. Based on the stage characterization study and the fact that an SpTCSeq includes a large amount of independent SpTCs (accounting for 91% of all SpTCs in a chemistry application as discussed in Section 2.2), we propose *stage parallelism* to improve hardware utilization for high performance. In particular, given an SpTCSeq, Athena co-runs an IOP-, FLOP-, or memory-intensive stage in an SpTC with alternative intensive stages in another SpTC.

Athena employs hyper-threading to co-run the stages with different intensive behaviors. This means that a memory-intensive stage and a compute (IOP/FLOP)-intensive stage or an IOP-intensive stage and a FLOP-intensive stage share a physical CPU core and use two hyperthreads to co-run. Because of the complementary characteristics of the two stages, using hyperthreading to co-run them increase instruction throughput (hence increasing CPU utilization). Athena co-runs two SpTCs but not more at the same time, because of the following reasons. (1) We conduct 32 tests using 12 input problems ranging from small to large datasets (see Table 3), and find that co-running two compute stages in a core using hyperthreading leads to at least 94.1% CPU utilization, which is sufficiently high; The co-run between an IOP-intensive stage and a FLOP-intensive stage is sufficient because of their compute-intensive feature. More than two SpTCs may incur instruction pipeline stall due to the limited integer or floating point function units. (2) Our tests also show that using one thread to run a memory-intensive stages consumes at least 60.3% of peak memory bandwidth. Hence, co-running a memory-intensive stage with another compute-intensive stage is enough to improve the utilization of memory bandwidth. In general, the accurate number of SpTCs to co-run is determined by the CPU utilization of individual stages and heavily relying on input and output data.

4 DATA MANAGEMENT ON PMM-BASED HETEROGENEOUS MEMORY SYSTEMS

We leverage the heterogeneous memory system to address the memory capacity bottleneck in an SpTCSeq.

4.1 Static Data Placement

We consider eight major data objects in the six stages of an individual SpTC. The eight major data objects are the two input tensors ($\mathcal{X}$ and $\mathcal{Y}$), the hash table-represented second input tensor (HtY), the thread-local hash table-based accumulator (HtA), the thread-local temporary data ($\mathcal{Z}_{local}$), the output tensor ($\mathcal{Z}_{pre}$) produced from previous SpTCs, the output tensor ($\mathcal{Z}$) in the current SpTC, and the hash table-represented output tensor (HtZ).

Athena uses the static data placement strategy [44] to decide the placement of $\mathcal{X}$, $\mathcal{Y}$, HtA, $\mathcal{Z}_{local}$ and $\mathcal{Z}$ on DRAM and PMM for individual SpTCs. This strategy considers memory access patterns associated with each data object, and places them in DRAM or PMM without migration in the middle of an SpTC execution. This strategy leads to higher performance than dynamic data placement, because of the avoidance of unnecessary data movement, discussed in [44]. In particular, for each SpTC, Athena places $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}_{pre}$ on PMM, because memory accesses to them are sequential and read-only in computation. Such a memory access pattern does not lead to big performance difference between placing data objects on DRAM and PMM, because of effective hardware prefetching and higher PMM performance in read (refer to [44] for details). Athena places HtA, $\mathcal{Z}_{local}$ and $\mathcal{Z}$ in DRAM, following the priority of $HtA > \mathcal{Z}_{local} > \mathcal{Z}$, according to the performance variance when moving them from PMM to DRAM (a data object causing higher variance has a higher priority). For large data objects such as HtA, $\mathcal{Z}_{local}$ and $\mathcal{Z}$, Athena makes the best efforts to place them on DRAM. This means that given a data object, if there is remaining DRAM space after excluding the memory consumed by data objects with higher priority, that data object is placed into DRAM as much as possible; If there is no remaining DRAM space, that data object is placed into PMM.

Athena is different from Sparta [44] in terms of data placement from the following perspectives. First, Athena manages data objects from all SpTCs together. This means that when the DRAM space is not large enough to save all data objects, not only data objects in an individual SpTC are managed following the priority discussed above, but also all data objects across SpTCs are managed following the above priority. This cross-SpTCs static data placement is feasible, because the sizes of data objects can be estimated [44] and the execution order of the six stages is known. For the data objects with the same priority in different SpTCs, Athena gives higher DRAM priority to those SpTCs with smaller memory footprint. This is because the SpTC with less memory footprint tends to have shorter execution time and hence can release the DRAM space to other SpTCs sooner.

Second, Athena dynamically migrates HtY and HtZ between DRAM and PMM, instead of using the static data placement in Sparta. This is because the two data objects have a large amount of random memory accesses. For example, the memory read/write accesses to HtY and HtZ account for 45% and 27% of all memory accesses in an SpTCSeq with the tensor Disilane (see Table 3 for Disilane). Placing them to DRAM can lead to significant performance improvement. However, the two data objects are the largest ones among all data objects and using the static data placement places most of data in PMM, which causes large performance loss. Athena uses a dynamic data placement strategy based on data semantics

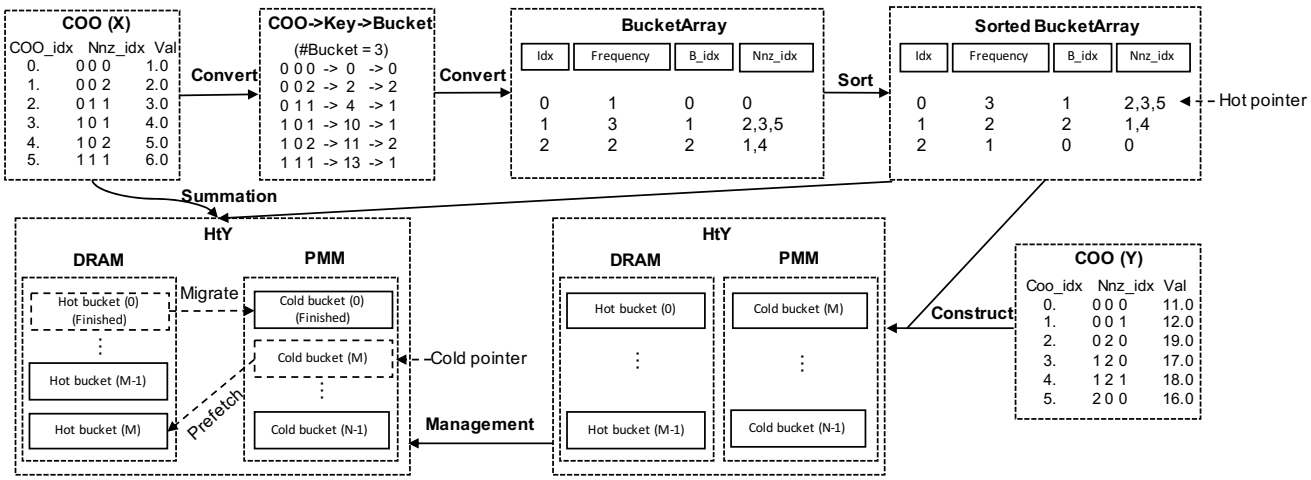

Figure 2: Workflow of the dynamic data placement based on data semantics.

to place hot data from the two data objects into DRAM as much as possible, discussed as follows.

4.2 Dynamic Data Placement based on Data Semantics

Dynamic data placement has been employed to enable high performance on heterogeneous memory [2, 12, 22, 24, 46, 57, 58, 61, 76, 78, 79, 82, 86, 87]. Most of those solutions are application agnostic, which means that they track page (or data) access frequency [2, 12, 22, 24, 78, 79, 82, 87] or manage DRAM as a hardware cache for PMM [46, 57, 76, 86] without the knowledge of data semantics. However, the data semantics gives critical indications on memory access patterns, which is useful to direct data placement and avoid unnecessary data movement. Leveraging data semantics to direct data placement has recently been used in data analytics workloads (e.g., traffic analysis) [64]. We study how to use data semantics to build HtY and HtZ and direct data placement in an SpTCSeq.

HtY and HtZ have random memory access patterns but still have hot non-zero elements frequently accessed. Those non-zero elements can be eliminated out of DRAM because of short-term variance in memory access patterns, if we use application-agnostic solutions. Using data semantics we can keep hot non-zero elements in DRAM to address the above problem. Furthermore, using data semantics allows us to know in advance which non-zero elements will be accessed, enabling effective prefetching from PMM to DRAM.

The existing HtY and HtZ built from $\mathcal{Y}$ and $\mathcal{Z}$ are based on the hash table [44], which is difficult to get the number of accesses for each element in advance to direct data placement, and the access order of non-zero elements in the hash table is also random, making prefetching difficult. Hence, we introduce a new method that exposes element hotness, during the construction of the hash table-based HtY and HtZ. As a result, using the semantics of HtY and HtZ, the data hotness is associated with data, allowing Athena to implement dynamic data placement and prefetching.

Figure 2 depicts the workflow of our design. Our design has four steps: bucket conversion, bucket sorting, hash table construction, and semantics-guided dynamic data placement.

Bucket conversion. Figure 2 uses tiny sparse tensors $\mathcal{X}$ and $\mathcal{Y}$ as an example. Using the method in [44], Athena first converts indices tuple of non-zero elements to keys based on the large-number representation function (LN), in order to make the key of each element unique. But different from [44], after the above conversion, Athena uses a common hash function (the Jenkins hash function) to distribute indices to different buckets. The number of buckets equals to the number of non-zero elements in $\mathcal{Y}$ (or $\mathcal{Z}$).

Bucket sorting. In the bucket conversion step, Athena records the number of non-zero elements in each bucket, which indicates the number of accesses to each bucket. The number of accesses to each bucket can be determined based on the number of non-zero elements, because SpTCSeq iterates non-zero elements in $\mathcal{X}$ (or $\mathcal{Z}_{pre}$) and then performs index search in HtY (or HtZ). The numbers of non-zero elements collected from buckets form a bucket array. The size of the bucket array is determined by the number of non-zero elements in $\mathcal{Y}$ (or $\mathcal{Z}$). Athena sorts the bucket array in an decreasing order. The sorting is necessary to enable quick identification of bucket hotness.

Hash table construction. Athena constructs the hash table-represented sparse tensor HtY from $\mathcal{Y}$ using the existing approach. But different form it, during the hash table construction, Athena traverses the sorted bucket array from the most accessed bucket (i.e., the bucket 0) and puts them into DRAM one by one till DRAM runs out of space. At that point, the remaining buckets, including those with the number of accesses as zero, are placed into PMM.

During the bucket placement on HM, Athena leverages a simple analytical model to estimate the memory requirement of each bucket: $Size_{idx} \cdot N_{\mathcal{X}} + Size_{val} + Size_{ep}$, in which $Size_{idx}$, $Size_{val}$ and $Size_{ep}$ are the size of an index, the size of a value, and the size of the entry pointer pointing to the bucket, respectively; $N_{\mathcal{X}}$ is the number of modes of $\mathcal{X}$.

Data-semantics guided data management. During the computation stages, Athena maintains two helper threads to manage

data between DRAM and PMM. The first helper thread is referred to as the *migration thread*. Whenever an element is accessed, the number of accesses for the corresponding bucket in the bucket array is reduced by one. Once the number of accesses for a hot bucket in DRAM becomes zero, meaning that the bucket will not be accessed any more, Athena put the hot bucket ID to an FIFO queue for the migration thread to move to PMM. The migration thread continuously checks the FIFO queue to migrate the bucket to PMM.

The second helper thread is referred to as the *prefetching thread*. When there is DRAM space for HtY (or HtZ) in DRAM, the prefetching thread migrates the hottest bucket from PMM to DRAM before it is needed by computation.

The semantics guided data management in Athena significantly improves the performance by directing data placement based on the expected data hotness/coldness using data semantics.

5 EVALUATION

5.1 Evaluation Setup

Platforms. We use an Intel Optane (PMM) Linux server, equipped with an Intel Xeon Cascade-Lake CPU including 24 physical cores at 2.3 GHz frequency. The CPU is attached with 6× 16 GB of DRAM and 6× 128 GB Intel PMM DIMMs. All implementations (Athena and other approaches) are compiled by gcc-7.5 and OpenMP 4.5 with `-O3`. All experiments were conducted on a single socket with one thread per physical core. Similar to recent work [24, 44, 78, 80], we use one-socket evaluation to highlight data movement between DRAM and PMM. Each workload is run 10 times and we report the average execution time.

Datasets We use sparse tensors summarized in Table 3. Those tensors are derived from real-world applications. Six tensors are derived from the well-known Coupled Cluster Singles and Doubles with perturbative triples correction, CCSD(T) [10] from chemistry [4]; Four tensors are derived from the notable Hubbard model from quantum physics in ITensor [17]; Two tensors are from large sparse tensor collection FROSTT [68]. Tensors in chemistry and physics are constructed by cutting off magnitude values smaller than 1×10^{-8} verified by domain scientists. We evaluate a real-world chemistry and four physics applications with Athena in Section 5.5 and Section 5.4 separately to study the effectiveness of Athena. We use a 4-SpTC sequence in Types 1 and 5 dependencies for each experiment to benchmark the performance if not mentioned otherwise. Section 5.5 will show the applications of chemistry using a real SpTCSeq with ten SpTCs. Eight tensors exceed the DRAM capacity (96 GB) on our platform, which indicates the necessity of using PMM.

Table 3: Characteristics of sparse tensors in the evaluation

Domains	Tensors	Order	Dimensions	#Non-zeros	Density
Chemistry	Benzene	4	336 × 336 × 42 × 42	4M	1.9×10^{-2}
	Cytosine	4	400 × 400 × 58 × 58	19M	3.4×10^{-2}
	Disilane	4	270 × 270 × 34 × 34	4M	4.2×10^{-2}
	Guanine	4	280 × 280 × 78 × 78	32M	6.6×10^{-2}
	Siosi3	4	64 × 64 × 186 × 186	6M	4.0×10^{-2}
	Uracil	4	90 × 90 × 174 × 174	10M	4.2×10^{-2}
Physics	Hubbard-1D-P	5	4 × 4 × 93 × 36 × 432	0.3M	6.3×10^{-3}
	Hubbard-1D-T	5	131 × 4 × 413 × 36 × 4	0.4M	5.1×10^{-3}
	Hubbard-1D-Z	5	4 × 129 × 184 × 24 × 4	0.1M	5.2×10^{-3}
	Hubbard-2D	5	4 × 4 × 111 × 24 × 528	0.3M	6.6×10^{-3}
Others	NIPS	4	2K × 3K × 14K × 17K	3M	1.8×10^{-6}
	Vast	5	165K × 11K × 2 × 100 × 89	26M	8.0×10^{-7}

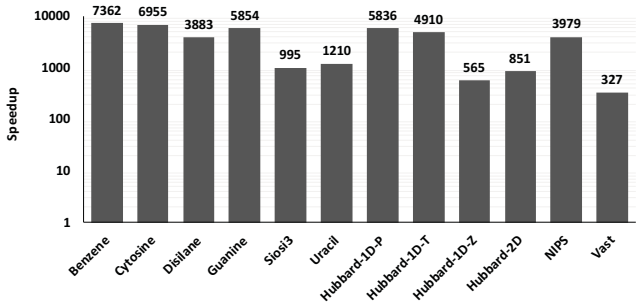

Figure 3: Overall speedups of Athena over Sparta for SpTC-Seq on 12 tensors.

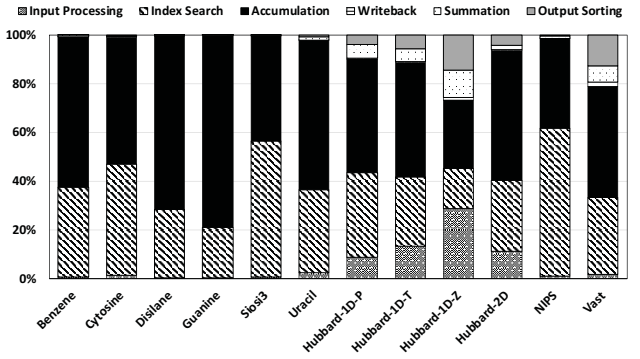

Figure 4: Percentage of execution time breakdown of Athena.

5.2 Overall Performance

In this experiment, to study the performance of Athena, we compare Athena with Sparta [44], the state-of-the-art element-wise sparse tensor contraction framework for an individual SpTC on heterogeneous memory. In general, as shown in Figure 3, Athena achieves 327-7362 × speedups over Sparta for SpTC sequences on 12 real-world tensors. Hash table-based sparse tensor summation contributes the most, 42-838 ×; while shared sparse accumulator and stage parallelism methods obtains 2.67-6.82 × and 1.21-1.46× speedup respectively. Performance analysis for every proposed optimization will be given in Section 5.3.

Figure 4 depicts the performance breakdown of Athena. Index search and accumulation stages are the most expensive stages for most tensors, which are in the computation part of an SpTC. Some tensors (e.g., Vast and Nell2) spend more time in output sorting than input processing stage, while some tensors (e.g., Hubbard-1D-T, Hubbard-1D-Z, Hubbard-2D) are vice versa, though they both use sorting and permutation algorithms. This is determined by the output tensor size versus the input tensors. For example, the size of the output tensor in Vast is 21× larger than the size of input tensor, while the size of the output tensor in Hubbard-1D-Z is 78% of the input tensor.

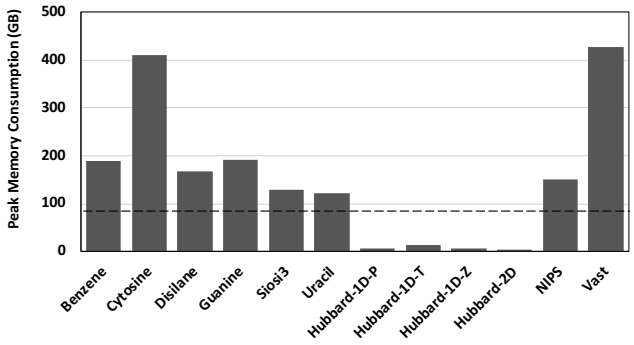

Figure 5: Peak memory consumption of SpTCSeq on 12 tensors.

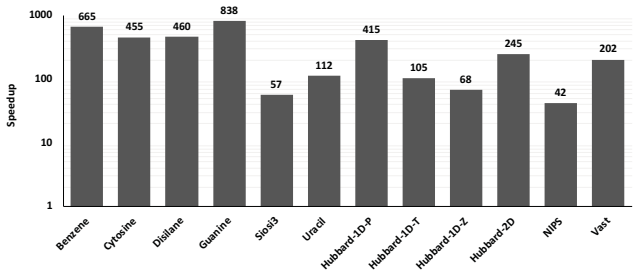

Figure 6: Speedups of Athena with hash-table represented summation over Sparta with traditional linear search-based summation.

Figure 5 shows the peak memory consumption of SpTC sequences in the experiment. Eight tensors consume more than DRAM space (96 GB), which cannot be performed without PMM memory. This indicates the large data used in applications and the necessity of using PMM. For even larger problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to the usage of the large PMM capacity.

5.3 Optimization Analysis

Hash table-based sparse tensor summation. Figure 6 shows the performance of using hash table-based sparse tensor summation on the 12 tensors respectively. In Figure 6, we observe that Athena significantly outperforms Sparta by 42-838 ×. The results show that our proposed hash table-based sparse tensor summation in Athena is more efficient than the traditional linear search-based summation.

Shared sparse accumulator. The shared sparse accumulator design in Athena reuses intermediate results of an SpTCSeq in Type 1 expression dependency to avoid redundant computation and memory operations and retains shared data objects to eliminate unnecessary input processing and data migration. Figure 7 shows the performance of using the shared sparse accumulator design ("Shared-*HtA*" in gray bars) in Athena compared to the sequential execution of the 4-SpTC sequence. We observe that Athena with the shared sparse accumulator design greatly outperforms the sequential execution by 2.67-6.82 ×, where Siosi3 obtains 6.82× and Hubbard-1D-Z is 2.67×. Because the performance improvement of shared sparse accumulator derives from eliminating the redundant computation and memory operations in some stages, the performance improvement of leveraging shared sparse accumulator depends on the weights of those stages (i.e., for the first SpTC, the process of intermediate results appending in the accumulation stage, writeback stage and output sorting stage; for the second SpTC, the process of input permutation/sorting in the input processing stage and the process of large-number conversion in the index search stage). For example, those stages account for 85.3% of the total execution time in the Siosi3 while only account for 62.5% of the total execution time in the Hubbard-1D-Z.

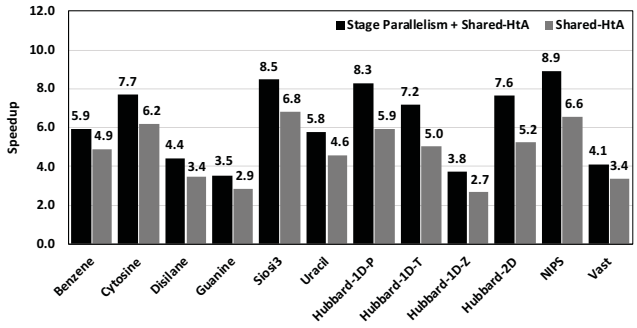

Figure 7: "Stage Parallelism" and "Shared-HtA" optimization speedup over the "Sparta + Summation" as the baseline.

Stage parallelism. Given a 4-SpTC sequence, the stage parallelism design in Athena co-runs stages in diverse patterns, IOP-, FLOP- and memory-intensive between two consecutive independent SpTCs. Figure 7 shows the performance of using the stage parallelism in Athena compared to its sequential execution for this SpTCSeq. We observe that our proposed stage parallelism outperforms the sequential execution using Athena with 21%-46% performance improvement. Athena with the stage parallelism improves 14-19% CPU utilization and 12-24% memory bandwidth compared to the sequential execution. The performance improvement in different tensors varies because the execution time of overlapped stages varies. For example, the stage parallelism gains 17.6% performance improvement on Vast while 31.5% on Disilane. Assume the ideal case without considering the potential resource contention of co-running a 4-SpTC sequence, the upper bound of performance improvement in Vast could achieve 21.2% and in Disilane is 37.8%. Our stage parallelism is quite close to ideal upper bound. The performance of the ideal case is measured by separately running stages in the critical path. For some small sparse input tensors, the thread scalability is poor due to the inadequate parallelism in the index search and accumulation stages. Stage parallelism can bring extra performance improvement in this case. For example, stage parallelism for the four small tensors in physics, having the least non-zeros in all 12 tensors, brings 11% to 22% extra performance improvement than the other eight.

Data management on PMM-based heterogeneous memory systems. We study the performance of employing the semantics guided data management on HM, compared with a state-of-the-art solution for HM management (i.e., IAL (Improved Active List) [83]), the hardware-managed cache approach (i,e, PMM Memory Mode),

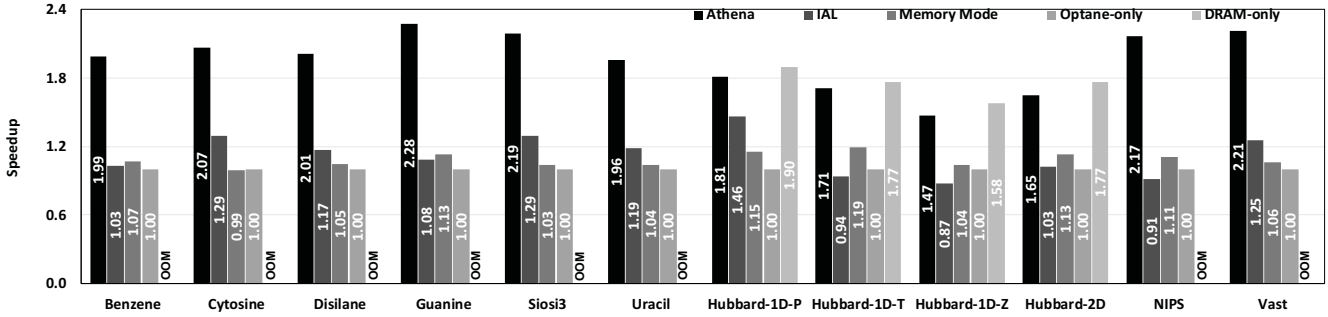

Figure 8: Speedups of Athena, IAL, Memory Mode and DRAM-only over PMM-only for SpTCSeq.

PMM-only (i.e., the AppDirect mode assigning all data objects to PMM) and DRAM-only (i.e., assign all data objects to DRAM). IAL is configured with its best configurations based on the IAL repository [84].

As shown in Figure 8, Athena with the semantics guided data management design outperforms IAL by 1.58× on average (up to 2.09×). Also, Athena achieves 1.82× (up to 2.58×) and 2.34× (up to 2.94×) performance improvement on average than PMM Memory Mode and PMM-only respectively. For some tensors (e.g., Hubbard-1D-Z), because the average memory bandwidth requirement is relatively smaller compared to others, the performance difference between Athena and PMM-only is small (47% improvement). For example, with Hubbard-1D-Z, if we place all data objects to DRAM (i.e., DRAM-only), the performance improvement is only 58%, compared to PMM-only.

We observe that the average PMM memory bandwidth of IAL is larger than that of Athena. This is because IAL causes undesirable data movement that consumes higher PMM memory bandwidth. The average DRAM memory bandwidth of PMM memory mode is larger than that of Athena, because PMM Memory Mode manages DRAM as a hardware cache for PMM and unnecessarily prefetches data objects to DRAM for high performance without being able to be aware of semantic hotness of data objects.

5.4 Performance Comparison to ITensor

In this experiment, we compare the performance of Athena and ITensor, which is a state-of-the-art library for block-sparse, multi-threading tensor contraction on a single machine. As applications in ITensor only include independent SpTCs (Type 5) without summation, we employ stage parallelism and semantic-hotspot-based

Table 4: A 10-SpTC sequence from a CCSD(T) model.

$K[h4,h3,h1,h2] += -0.125 * L[p1,p2,h3,h4] * M[p1,p2,h1,h2]$
$N[p3,p4,h1,h2] += 1.0 * K[h4,h3,h1,h2] * M[p3,p4,h4,h3]$
$O[p1,h3,p4,h2] += 0.5 * L[p2,p4,h3,h1] * M[p1,p2,h1,h2]$
$N[p3,p4,h1,h2] += 1.0 * O[p4,h4,p1,h1] * M[p3,p1,h4,h2]$
$P[p1,h3,p4,h2] = -0.5 * L[p2,p4,h3,h1] * Q[p2,p1,h1,h2]$
$R[p3,p4,h1,h2] += -1.0 * P[p4,h4,p1,h1] * Q[p1,p3,h4,h2]$
$S[p1,h3,p4,h2] += 0.5 * T[p2,p4,h3,h1] * U[p1,p2,h1,h2]$
$V[p3,p4,h2,h1] += 1.0 * S[p4,h4,p1,h1] * Q[p3,p1,h2,h4]$
$R[p3,p4,h1,h2] += 1.0 * S[p4,h4,p1,h1] * U[p3,p1,h4,h2]$
$V[p3,p4,h2,h1] += -1.0 * P[p4,h4,p1,h1] * M[p3,p1,h4,h2]$

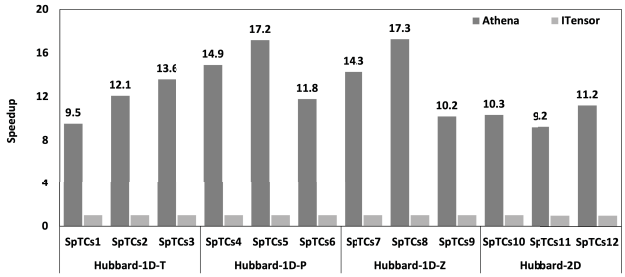

Figure 9: Speedups of Athena over ITensor on Hubbard-1D-T, Hubbard-1D-P, Hubbard-1D-Z and Hubbard-2D models using different SpTCSeq with different sparse input tensors.

data management in Athena and compare the performance of Athena and with ITensor. SpTCs with different tensors (SpTCs1 to SpTCs12) are from well-known quantum physics models (Hubbard-1D-T, Hubbard-1D-P, Hubbard-1D-Z and Hubbard-2D) [15] in ITensor [16] with cutting off values smaller than 1×10^{-8}. Figure 9 shows the performance comparison between Athena and ITensor. We observe that Athena significantly outperforms ITensor with 12.6× performance improvement on average.

5.5 Application in Chemistry

We study the performance of Athena on a real-world SpTC sequence from NWChem in chemistry. NWChem is a well-known computational chemistry library for quantum chemical and molecular dynamics functionality [4]. We select a 10-SpTC sequence derived from CCSD(T) [10]. The 10-SpTC sequence is concluded in Table 4 and cover 5 different expression types. We compare the performance of Athena to Sparta [44] on this sequence. Athena achieves 6232× speedup over Sparta combining all our designs. In particular, Athena achieves 635× speedup with hash table-based sparse tensor summation; 1.9× with semantic-hotspot-based data management; 4.3× with shared sparse accumulator; 1.2× with stage parallelism.

6 RELATED WORK

Sparse tensor contraction. Dense tensor contraction has been studied for decades on diverse hardware platforms [5, 19, 21, 27, 28, 32, 34, 42, 50, 65, 72, 73], in scientific computing including chemistry, physics, and mechanics. The state-of-the-art studies focus on block-sparse tensor contractions with dense blocks in tensors. The

conventional approaches first extract dense block-pairs of the two input tensors, and then perform multiplication by calling dense BLAS linear algebra. Finally, those approaches pre-allocate the output tensor using domain knowledge or a symbolic phase approach [20, 25, 53, 54, 67], such as TiledArray [54], Cyclops Tensor Framework [36], and libtensor [14, 49]. The state-of-the-art work Sparta focuses on element-wise sparse tensor contractions [44], solving the high dimensionality challenges through hash table-based approaches and addressing the unknown output tensor and irregular memory access challenges by dynamic allocation, permutation and sorting. Athena develops element-wise sparse tensor contraction by optimizing tensor summation as well, frequently occurred in contraction sequences.

Sparse tensor contraction sequences. Sparse tensor contraction often occurs as sequences in a spectrum of applications, such as quantum chemistry, quantum physics and deep learning [17, 31, 41, 62, 63]. Some existing work optimizes tensor computation sequences. AutoHOOT [48] decomposes a dense tensor contraction workload into task sequences and overlaps the computation and communication task sequences to reduce the communication overhead in a distributed execution. DLTC [34] takes input tensor computation sequences and generates optimized derivative sequences by automatic differentiation. TensorFlow [1] leverages a directed acyclic graph to represent the computation and data flow of tensor-based operator sequences and co-run the tensor-based operator sequences in an FIFO method. Athena is different from them in terms of leveraging the domain-knowledge of SpTC sequences to achieve high performance.

Data management on heterogeneous memory systems. Heterogeneous memory management attracted plenty of research efforts in recent years [2, 22, 24, 59, 60, 81, 82]. These works explore various page-level data placement polices on HM based on main memory access profiling result. Thermostat [2] uses sampling-based profiling to track page table and migrates hot pages into DRAM. RAMinate [22], Heteros [24], Yan et al. [82] propose the state-of-the-art memory management solutions for general purpose which guides page placement based on an existing Linux page replacement mechanism. Application-specific HM management solutions [8, 11, 13, 18, 33, 35, 45, 47, 75, 78, 79, 85] leverage domain knowledge to further improve performance. MyNVM [13] proposes a software-managed multi-level caches policy to treat DRAM and NVM as caches for hard drives. Sparta [44] leverages application awareness and static data placement to avoid unnecessary data movement. Athena is different from these works in terms of exposing data semantics and dynamically managing data objects across SpTCs.

7 CONCLUSION

Efficiently computing sparse tensor contraction sequences (SpTCSeq) is critical to many applications. However, it is challenging, due to its redundant computation and memory operations, massive memory consumption, and inefficient utilization of hardware. In this paper, we explore solutions to address those challenges based on algorithm knowledge and characterization of workloads in SpTCSeq. We introduce Athena, a high performance framework for SpTC sequences. Athena is based on a set of novelty in data structures, runtime techniques, and emerging Optane-based memory architecture. Evaluating with 12 datasets, we show that Athena brings significant speedup (327-7362 ×) over the state-of-the-art SpTC algorithm. Athena also showcases its effectiveness in quantum chemistry and physics applications. For exascale problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to its capability to solve large sparse tensors on each single node.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their constructive comments. This work was partially supported by U.S. National Science Foundation (CNS-1617967, CCF-1553645 and CCF-1718194), the Chameleon Cloud, and Intel hardware donation. This research was 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] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[2] 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.

[3] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M. Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *J. Mach. Learn. Res.*, 15(1):2773–2832, January 2014.

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

[7] 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.

[8] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, 2020.

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

[10] T Daniel Crawford and Henry F Schaefer. An introduction to coupled cluster theory for computational chemists. *Reviews in computational chemistry*, 14:33–136, 2000.

[11] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2021.

[12] 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.
[13] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.
[14] 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.
[15] Tilman Esslinger. Fermi-hubbard physics with atoms in an optical lattice. *Annu. Rev. Condens. Matter Phys.*, 1(1):129–152, 2010.
[16] 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.
[17] Matthew Fishman, Steven R White, and E Miles Stoudenmire. The ITensor software library for tensor network calculations. *arXiv preprint arXiv:2007.14822*, 2020.
[18] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory, 2019.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] Joyce C. Ho, Joydeep Ghosh, and Jimeng Sun. Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 115–124, New York, NY, USA, 2014. ACM.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] T. Kolda and B. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.
[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] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.
[34] 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.
[35] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019.
[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] 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.
[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, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.
[45] Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. {RIANN}: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 {USENIX} Conference on Operational Machine Learning (OpML 20)*, 2020.
[46] 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.
[47] Jie Liu, Jiawen Liu, Wan Du, and Dong Li. Performance analysis and characterization of training deep learning models on mobile device. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 506–515. IEEE, 2019.
[48] Linjian Ma, Jiayu Ye, and Edgar Solomonik. Autohoot: Automatic high-order optimization for tensors. *arXiv preprint arXiv:2005.04540*, 2020.
[49] 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.
[50] Devin Matthews. High-performance tensor contraction without BLAS. *CoRR*, abs/1607.00291, 2016.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] 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.

[56] Christos Psarras, Lars Karlsson, and Paolo Bientinesi. The landscape of software for tensor computations. *arXiv preprint arXiv:2103.13756*, 2021.

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

[58] 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021.

[59] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference (ATC)*, 2021.

[60] Jie Ren, Kai Wu, and Dong Li. Exploring non-volatility of non-volatile memory for high performance computing under failures. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 237–247. IEEE, 2020.

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

[62] 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.

[63] 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.

[64] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2020.

[65] 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.

[66] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on Signal Processing*, 65(13):3551–3582, July 2017.

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

[68] 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.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019.

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

[77] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. University of California, Berkeley Berkeley, CA, 2008.

[78] 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.

[79] 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.

[80] 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.

[81] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: Memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the 35th ACM International Conference on Supercomputing*, 2021.

[82] 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.

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

[84] 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.

[85] 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.

[86] 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.

[87] 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.