Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 17-006

Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory

Shaden Smith, Jongsoo Park, George Karypis

June 5, 2017

Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory

Shaden Smith*, Jongsoo Park†, George Karypis*
* Department of Computer Science and Engineering, University of Minnesota
† Parallel Computing Lab, Intel Corporation
{shaden, karypis}@cs.umn.edu, jongsoo.park@intel.com

***Abstract*—HPC systems are increasingly used for data intensive computations which exhibit irregular memory accesses, non-uniform work distributions, large memory footprints, and high memory bandwidth demands. To address these challenging demands, HPC systems are turning to many-core architectures that feature a large number of energy-efficient cores backed by high-bandwidth memory. These features are exemplified in Intel's recent Knights Landing many-core processor (KNL), which typically has 68 cores and 16GB of on-package multi-channel DRAM (MCDRAM). This work investigates how the novel architectural features offered by KNL can be used in the context of decomposing sparse, unstructured tensors using the canonical polyadic decomposition (CPD). The CPD is used extensively to analyze large multi-way datasets arising in various areas including precision healthcare, cybersecurity, and e-commerce. Towards this end, we (i) develop problem decompositions for the CPD which are amenable to hundreds of concurrent threads while maintaining load balance and low synchronization costs; and (ii) explore the utilization of architectural features such as MCDRAM. Using one KNL processor, our algorithm achieves up to $1.8\times$ speedup over a dual socket Intel Xeon system with 44 cores.**

I. Introduction

HPC systems are increasingly used for data intensive workloads, such as those that arise in many machine learning applications. These applications process data coming from sources such as electronic health records, social networks, and retail. These workloads are highly unstructured and commonly exhibit irregular memory accesses, non-uniform data distributions, large memory footprints, and demand high memory bandwidth.

The growing demands of data intensive applications have driven the adoption of *many-core processors*. These are highly parallel processors that feature several tens of cores with wide vector instructions. The large processing capabilities of many-core processors places increased pressure on memory bandwidth that DDR memory is unable to satisfy. A key recent trend of many-core processors is the inclusion of high-bandwidth memory, which has several times higher bandwidth than traditional DDR memory. This trend is exemplified in recent architectures such as NVIDIA Pascal, AMD Fiji, and Intel Knights Landing (KNL).

In order to harness the power of many-core processors, applications must expose a high degree of parallelism, load balance tens to hundreds of parallel threads, and effectively utilize the high-bandwidth memory. Algorithm developers must embrace these challenges and, in many cases, re-evaluate existing parallel algorithms.

To that end, we present an exploration of performance optimizations for a characteristic data intensive workload. We study the KNL many-core processor and use the canonical polyadic decomposition (CPD) [1] as a benchmark.

Tensors are the generalization of matrices to more than two dimensions (called *modes*). The CPD is a commonly used approach to approximate a tensor as the summation of a small number of rank-one tensors. The CPD is an important tool in areas such as healthcare [2], cybersecurity [3], and recommender systems [4]. Sparse tensor factorization is of particular interest to our study because it features many of the classic challenges present in sparse matrix kernels, while bringing new challenges that arise with multi-way data.

We identify and address challenges that involve both the algorithms used for the CPD and the utilization of KNL's architectural features. Specifically, we address issues associated with extracting concurrency at different levels of granularity from unstructured and sparse data, load balancing computations in the presence of highly skewed data distributions, and understanding the different trade-offs between fine-grained synchronization and storage overheads. Our contributions include:

1) Algorithmic improvements to increase the degree of concurrency, decrease synchronization overheads, and improve load balance for hundreds of threads.
2) A strategy for explicitly managing high-bandwidth memory to achieve high performance when the dataset exceeds its capacity.
3) An extensive experimental evaluation on a variety of real-world datasets that demonstrates the effectiveness of both the algorithmic improvements and KNL's hardware features.
4) A comparison of hardware atomics on traditional and many-core processors, and recommendations for future many-core architectures.

The rest of this paper is organized as follows. Section II provides an overview of the KNL architecture and a background on tensor factorization. Section III reviews related

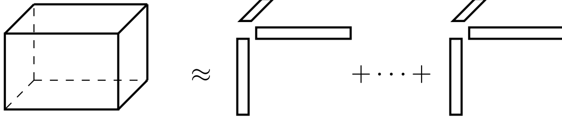


Figure 1: The canonical polyadic decomposition as a sum of outer products.

work on parallel sparse tensor factorization. Section IV details our algorithm for computing the CPD on a KNL processor. Section V presents experimental results. Lastly, Section VI offers concluding remarks.

II. Preliminaries

A. Knights Landing (KNL)

KNL [5] is the second generation Xeon Phi many-core processor from Intel. KNL has up to 72 cores, each with two 512-bit vector processing units and 4-way simultaneous multi-threading. Cores have 32KB of L1 cache and are arranged in pairs which share 1MB of L2 cache.

KNL includes two types of memory: DDR4 and multi-channel DRAM (MCDRAM). MCDRAM offers up to 480GB/s achievable memory bandwidth measured by the STREAM benchmark [6], or approximately $4\times$ the bandwidth of the latest 2-socket Xeon systems with DDR4. MCDRAM can be configured to either be explicitly managed by the software (*flat mode*), used as a last-level cache (*cache mode*), or a combination of the two (*hybrid mode*).

B. Notation

Tensors are denoted using bold calligraphic letters ($\boldsymbol{\mathcal{X}}$) and matrices using bold letters ($\mathbf{A}$). Tensors have M modes of lengths $I_1, \ldots, I_M$ and $\text{nnz}(\boldsymbol{\mathcal{X}})$ non-zeros. Entries in tensors and matrices are denoted $\boldsymbol{\mathcal{X}}(i_1, \ldots, i_M)$ and $\mathbf{A}(i, j)$, respectively. A colon in the place of an index takes the place of all non-zero entries. For example, $\mathbf{A}(i, :)$ is the ith row of $\mathbf{A}$. A *fiber* is a vector produced from holding all but one index constant (e.g., $\boldsymbol{\mathcal{X}}(i_1, \ldots, i_{M-1}, :)$). An important matrix kernel is the *Hadamard product*, or element-wise product. The Hadamard product is denoted $\mathbf{A} \circledast \mathbf{B}$.

C. Canonical Polyadic Decomposition (CPD)

The CPD is one interpretation of the SVD in tensor algebra. Shown in Figure 1, the CPD models a tensor with a set of M matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I_1 \times F}, \ldots, \mathbf{A}^{(M)} \in \mathbb{R}^{I_M \times F}$, where F is the desired model rank. Applications in machine learning are usually interested in the low-rank CPD, where F is a small constant on the order of 10 or 100.

Computing the CPD is a non-convex optimization problem most often approximated with an iterative alternating least squares (CPD-ALS) [1]. The concept behind CPD-ALS is that by holding all but one factor constant, the solution has a least squares solution. An important consequence of using ALS is that computations are *mode-centric*, meaning each of the kernels will update one mode of the factorization at a time while reading all others. Therefore, when discussing algorithmic details we will sometimes only refer to the first mode and imply that the other modes proceed similarly.

Algorithm 1 CPD-ALS

1: **while** not converged **do**
2:     **for** $m = 1 \to M$ **do**
3:         $\mathbf{G} \leftarrow \circledast_{n \neq m} \mathbf{A}^{(n)T} \mathbf{A}^{(n)}$
4:         $\mathbf{Y} \leftarrow \text{MTTKRP}(\boldsymbol{\mathcal{X}}, \{\mathbf{A}^{(n)}\}_{n \neq m})$
5:         $\mathbf{A}^{(m)} \leftarrow \mathbf{Y}\mathbf{G}^{-1}$
6:     **end for**
7: **end while**

Algorithm 1 details the CPD-ALS algorithm. Line 3 constructs the Gram matrix of size $F \times F$. While each $\mathbf{A}^{(m)T}\mathbf{A}^{(m)}$ computation requires $\mathcal{O}(F^2 I_m)$ work, they can be cached and only require recomputation after $\mathbf{A}^{(m)}$ is updated. Line 4 performs a chain of $M-1$ tensor-vector products using F vectors, often referred to as the *matricized tensor times Khatri-Rao product* (MTTKRP).

MTTKRP is the most computationally expensive step of CPD-ALS. The output of MTTKRP for the mth mode is $\mathbf{Y} \in \mathbb{R}^{I_m \times F}$. Each non-zero $v = \boldsymbol{\mathcal{X}}(i_1, \ldots, i_M)$ contributes:

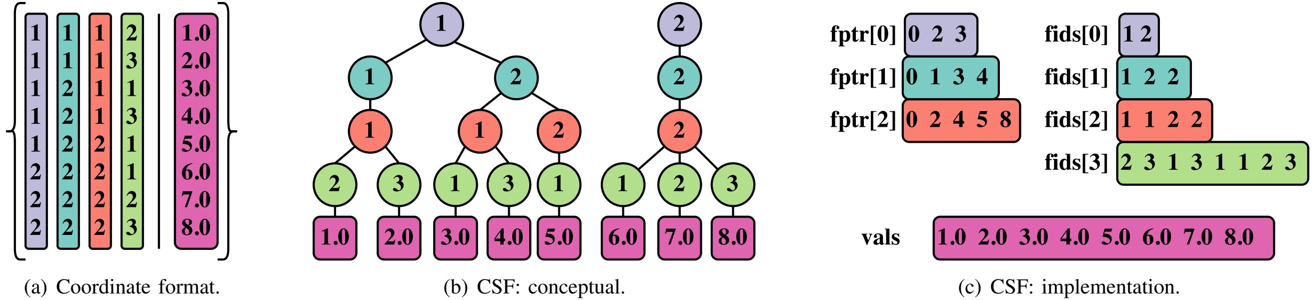
$$\mathbf{Y}(i_m, :) \leftarrow \mathbf{Y}(i_m, :) + v \left(\underset{n \neq m}{\circledast} \mathbf{A}^{(n)}(i_n, :) \right), \qquad (1)$$

for a total of $(M{+}1)F\,\text{nnz}(\boldsymbol{\mathcal{X}})$ operations.

D. Compressed Sparse Fiber (CSF)

In prior work, we proposed the *compressed sparse fiber* (CSF) data structure for sparse tensors [7], [8]. CSF is a generalization of the compressed sparse row (CSR) data structure that is popular for sparse matrices. Figure 2 illustrates a four-mode tensor in CSF format. CSF recursively compresses each mode of the tensor into a set of I_1 prefix trees. Each path from a root to leaf forms one tensor non-zero. CSF is implemented with three multidimensional arrays: (i) `fptr`, which encodes the sparsity structure; (ii) `fids`, which stores the label of each node; and (iii) `vals`, which stores the non-zero values. Conceptually, the `fptr` structure is a sequence of M `rowptr` arrays used in CSR. Each array is used to index into the next in the sequence, with the final array indexing directly into the non-zeros.

The tree structure reveals opportunities for computational and bandwidth savings. Consider a fiber of $\boldsymbol{\mathcal{X}}$, which is represented as a set of siblings in the leaf level of the tree. Their contributions to Equation 1 vary only in the non-zero values and the Hadamard products involving the final mode. This redundancy can be exploited by moving from an element-wise algorithm to a *fiber-centric* algorithm. The algorithm (i) accumulates the interactions of all non-zeros with the last factor matrix; (ii) scales the accumulation by the overlapping Hadamard products; and (iii) adds the result to $\mathbf{Y}(i_1, :)$. If a fiber has l non-zeros, this fiber-centric algorithm



(a) Coordinate format. (b) CSF: conceptual. (c) CSF: implementation.

Figure 2: Compressed Sparse Fiber. Each path from root to leaf in (b) encodes a non-zero found in (a).

eliminates $l-1$ Hadamard products and their corresponding $l-1$ accesses to matrix rows. Furthermore, this algorithm is recursive in that subtrees which have common ancestors can share the overlapping subset of Hadamard products.

Figure 2 shows the tensor modes in the user-supplied order. When storing a tensor in CSF, the ordering of the modes is arbitrary and any one of the $M!$ permutations is a valid option. However, the ordering can significantly affect the amount of required storage. An effective heuristic for achieving high levels of compression is to sort the modes by length, with the shortest mode at the top of the tree [8].

The above discussion focused on performing an MTTKRP operation with respect to the mode of $\boldsymbol{\mathcal{X}}$ stored at the root. A reformulation of the above algorithm permits one to perform all MTTKRP operations with an arbitrarily-ordered tensor representation [8]. Assume that we are operating with respect to the mode stored at the mth level of the CSF. The algorithm proceeds as a depth-first traversal on each tree in the CSF structure. As the traversal moves downwards toward the mth level, Hadamard products are computed and stored for later use. Next, starting from the leaves, non-zero contributions and their corresponding Hadamard products are computed and brought up the tree to the $(m+1)$th level. Finally, the contributions from above and below are joined with a final Hadamard product and added to $\mathbf{Y}(i_m,:)$. The only auxiliary memory required is an $M \times F$ matrix for accumulating and propagating Hadamard products.

III. Related Work

Heinecke et al. [9] explored optimizations in a seismic simulation code on KNL. They show $3\times$ improvement over the previous generation Xeon Phi by tuning small matrix-matrix multiplication kernels for AVX-512, managing MCDRAM by using an out-of-core computational kernel, and decreasing memory traffic by selectively replicating and scattering data. Sparse matrix-vector multiplication also has a history of optimization on many-core processors [10]–[12].

Several works have optimized sparse MTTKRP for shared- and distributed-memory systems. Baskaran et al. [13] proposed a NUMA-aware balanced work scheduler for sparse tensor operations. The scheduler is designed for a fine-grained element-wise algorithm. Ravindran et al. [14] presented an MTTKRP formulation for three-mode tensors that accesses only non-zeros by $\boldsymbol{\mathcal{X}}(:,i_2,i_3)$ slabs, and thus relies on a single tensor representation. Their formulation achieves the same computational savings as the generalized algorithm for CSF. Distributed-memory algorithms were developed by Choi and Vishwanathan [15], Kaya and Uçar [16], and Smith and Karypis [17].

IV. Many-Core Sparse Tensor Factorization

We now detail our method of obtaining high performance on KNL. This is a challenge which spans both high-level design and low-level implementation. The problem decomposition must expose a sufficient amount of parallelism, load balance hundreds of threads, and minimize fine-grained synchronization. Additionally, the implementation must utilize advanced hardware features such as vector instructions, efficient synchronization primitives, and MCDRAM.

A. Problem Decomposition for Many-Core Processors

1) Partial Tensor Tiling: The existing CSF-based algorithms use coarse-grained parallelism via distributing individual trees to threads. Computing with respect to the root mode has no race conditions to consider, as each root node ID is unique. There are no uniqueness guarantees for levels below the root, and thus we must consider the case of threads overlapping additions to $\mathbf{Y}(i_m,:)$, where m is a level below the root. Two solutions were proposed [8]: a mutex pool can be indexed by node IDs to protect rows during updates; or $\boldsymbol{\mathcal{X}}$ can be tiled using a grid of dimension P^M, where P is the number of threads. Note that root nodes are no longer unique if tiling is used, and thus it must be performed on all M modes. Tiling for $P{=}2$ is illustrated in Figure 3. Expensive synchronization is avoided by distributing the mode-m layers of tiles to threads.

This approach of decomposing the computations is limited in two major ways. First, coarse-grained parallelism is only effective when the tensor modes are sufficiently long. Many real-world tensors exhibit a combination of long, sparse

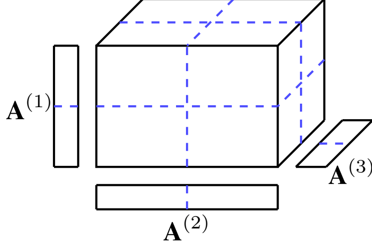


Figure 3: Tiling for two threads to avoid synchronization. Layers of mode-m blocks are distributed to threads.

modes and short, substantially more dense ones. For example, tensors used in context-aware recommendation will have many users but only a few contexts (e.g., time or location of purchase). Indeed, the recent performance evaluation by Rolinger et al. [18] showed that CSF-based computation is severely impacted by tensors with short modes. Second, tiling each tensor mode to avoid synchronization faces serious scalability issues when there are many threads or many tensor modes. For example, a six-mode tensor on a 68-core KNL system would require $68^6 \approx 99$ billion tiles. The alternative of relying solely on a mutex pool performs poorly even at small thread counts [8].

To address these problems, we developed a method that tiles a subset of the tensor modes and uses atomics for the remaining ones. In determining which modes to tile, priority is given to the longest modes for two reasons. First, they provide more opportunities for parallelism by permitting us to decompose the long tensor modes. As long as the tensor has at least one mode which is sufficiently long to distribute among P threads, then this decomposition is not hindered by the presence of short modes. Second, the longer modes contain more nodes in the CSF structure than shorter ones, and thus require more frequent synchronization. This intuition is supported in a previous evaluation which showed that the last modes (i.e., the lower levels of the CSF structure) experienced more overhead from fine-grained synchronization than the first modes [8].

Tiling long modes fits easily into the CSF framework because the modes are sorted by length. Our partial tiling is parameterized by h, referred to as the *tiling height*. The tiling height which defines a level in the CSF data structure and also how many modes will be tiled. When we compute for a mode which resides above level h, atomics are used to prevent race conditions. At or below level h, tile layers are distributed to threads to avoid using atomics. The resulting tensor will have P^h tiles, where P is the number of threads being used. For example, h=0 results in an untiled tensor, h=1 results in a 1D decomposition on the longest mode, and h=M tiles each mode.

Our method of partial tiling bears some resemblance to the medium-grained decomposition for distributed-memory MTTKRP [17]. The medium-grained decomposition imposes a $(q \times r \times s) = P$ grid over the tensor to distribute non-zeros and balance communication volume. The two decompositions vary in that partial tiling does not require exactly P blocks of non-zeros. Furthermore, if a mode is tiled its layers will be distributed among threads instead of individual blocks of non-zeros.

2) Multiple Tensor Representations: A single CSF representation is attractive because of its low memory overhead compared to M specialized representations [8]. However, beyond the previously discussed synchronization challenges, an additional disadvantage of using a single CSF representation is less favorable writes to memory. Consider the difference between computing with respect to the first and Mth mode. During the first mode, $\mathbf{Y}$ is written to once at the conclusion of each tree and $\mathbf{A}^{(M)}$ is read for each non-zero (see Section II-D). In contrast, updating the Mth mode involves reading from $\mathbf{A}^{(1)}$ once at the start of each tree and updating $\mathbf{Y}$ for each non-zero. Writing for each non-zero places significantly more pressure on memory bandwidth and cache coherency mechanisms. Furthermore, the updates to $\mathbf{Y}$ follow the sparsity pattern and are generally scattered, which challenges the hardware prefetcher. Additionally, when $\mathbf{Y}$ has few columns, only a few bytes out of a cache line may be utilized.

We propose to use two CSF representations when memory allows. The first CSF is as before, with the modes sorted and the shortest mode at the root level. The second CSF places the longest mode at the root level and sorts the remaining modes by non-decreasing length. When computing for the longest mode, we use the second CSF in order to improve access patterns. Since the longest mode is placed at the root level, we forego tiling on the second CSF in order to avoid any additional storage or computational overheads.

Note that this concept is not limited to one, two, or M representations. However, the combinations of CSF representations and orderings grows exponentially. Due to the combinatorial nature of the problem, we restrict our focus to either one, two, or M representations; each with modes sorted by length except for the specialized root.

3) Load Balancing the Computations: Many tensors have a non-uniform distribution of non-zeros. Algorithms which rely on distributing whole slices or tiles to many threads (i.e., a coarse-grained decomposition) can exhibit severe load imbalance. On the other hand, a fine-grained decomposition which distributes individual non-zeros can load balance a computation at the cost of frequent synchronizations.

We refer to a slice with a disproportionately large number of non-zeros as a *hub* slice. We call slice i a hub slice if

$$\text{nnz}(\boldsymbol{\mathcal{X}}(i,:,\ldots,:)) \geq \delta \left(\frac{\text{nnz}(\boldsymbol{\mathcal{X}})}{P} \right),$$

where δ is a user-supplied threshold. We empirically found δ=0.5 to be an effective value.

When a hub slice is identified during the construction of the CSF, it is not assigned to a thread. Instead, we evenly distribute all of its non-zeros among threads as a form of fine-grained parallelism. During the MTTKRP operation, all threads first process the set of non-hub slices in parallel using any synchronization construct as before. Second, each thread processes its assigned portion of the hub slices. By definition, there cannot be many hub slices relative to the total number of slices, and thus synchronization overheads are negligible.

B. Leveraging Architectural Features

1) Vectorization: KNL (and other modern architectures) heavily rely on vectorization to achieve peak performance. KNL uses the new AVX-512 instruction set and has two 512-bit vector units per core. The MTTKRP formulation that we use accesses the tall, skinny factor matrices in a row-major fashion. Processing a node in the CSF structure requires $\mathcal{O}(F)$ operations. The micro-kernels are either in the form of (i) scaling a row by a non-zero value (i.e., a BLAS-1 *axpy*), or (ii) an element-wise multiplication of two rows (i.e., a Hadamard product). In practice, useful values of F will saturate at least one vector width. Therefore, we vectorize each of the micro-kernels.

2) Synchronization: Partial tiling of the tensor requires synchronization of some form to be used on the untiled tensor modes. The choice of synchronization primitive is heavily dependent on both hardware support and the characteristics of the data (e.g., whether conflicts are expected to be common). We will now discuss several options and evaluate them in Section V-D.

Mutexes: The most simple synchronization strategy, and the one used in prior work [8], is to surround updates with a mutex. Most writes will be to unique data when the tensor mode is sufficiently large. We can maintain a pool of mutexes in order to reduce lock contention, but performance is still limited when mutexes have a high overhead in hardware. A challenge of mutexes is that we must tune the size of the pool to find the best trade-offs between storage overhead and contention.

Compare-and-Swap: CAS instructions are often optimized in hardware and bring no storage overhead unlike a mutex pool. Their limitation comes from the granularity of the protected memory. CAS is currently limited to 16 bytes on KNL and other popular architectures. Thus, four CAS instructions must be issued to utilize a full cache line (or full vector register) on a KNL system.

Transactional Memory: Modern architectures such as Intel's Haswell and Broadwell include hardware support for restricted transactional memory (RTM). While KNL does not include RTM, it is an efficient option for other parallel architectures.

Privatization: Hardware atomics may introduce a large overhead when the tensor mode is small and contention is probable. In that event, we instead allocate a thread-local matrix which is the same size as $\mathbf{Y}$. Each thread accumulates into its own buffer without need for atomics. Finally, all buffers are combined with a parallel reduction. The memory overhead of privatization makes it only practical for short modes. We privatize mode m if

$$I_m P \leq \gamma \, \mathrm{nnz}(\boldsymbol{\mathcal{X}}), \tag{2}$$

where P is the number of threads and γ is a user-supplied threshold. We empirically found γ=0.2 to be effective.

3) Managing High-Bandwidth Memory: The 16GB capacity of MCDRAM on KNL is sufficient to factor some, but not all tensors. When the working set entirely fits in memory, explicitly managing MCDRAM and running in cache mode should offer similar performance.

When the working set exceeds the MCDRAM capacity, we prioritize placement of the factor matrices in MCDRAM. Each node in the CSF structure consumes $\mathcal{O}(1)$ memory but spawns $\mathcal{O}(F)$ accesses to the factors. In total, the CSF structure consumes $\mathcal{O}(\mathrm{nnz}(\boldsymbol{\mathcal{X}}))$ bandwidth. When the tensor and factors exceed the size of MCDRAM, it is likely that the the factors do not fit in the on-chip caches and thus consume $\mathcal{O}(F \cdot \mathrm{nnz}(\boldsymbol{\mathcal{X}}))$ bandwidth.

V. Experimental Methodology & Results

A. Experimental Setup

We use two hardware configurations for experimentation. One machine has two sockets of 22-core Intel Xeon E5-2699v4 Broadwell processors, each with 55MB of last-level cache, and 128GB of DDR4 memory. The second machine has an Intel Xeon Phi Knights Landing 7250 processor with 68 cores, 16GB of MCDRAM, and 94GB of DDR4 memory. Throughout our discussion, we will refer to the dual-socket Broadwell machine as BDW and the Knights Landing machine as KNL. Importantly, KNL is a socketed processor and all application code runs directly on the hardware. Thus, there are no PCIe transfer overheads to consider. Unless otherwise specified, KNL is configured in flat mode with quadrant configuration.

Source code is written in C++ and modified from SPLATT v1.1.1, a library for sparse tensor factorization [19]. We use double-precision floating point numbers, 64-bit integers for indexing non-zeros, and 32-bit integers for node IDs. The MTTKRP kernel is optimized with both AVX2 intrinsics for BDW and AVX-512 intrinsics for KNL. We use the Intel compiler version 17.0.0 with `-xCORE-AVX2` on BDW and `-xMIC-AVX512` on KNL, and Intel MKL for LAPACK routines. All source code is publicly available[1]. We set the environment variable `KMP_LOCK_KIND=tas` to use test-and-set locks [20].

Reported runtimes are the arithmetic mean of thirty iterations and error bars mark the standard deviation. Unless otherwise noted, we use F=16 for experiments.

[1] http://cs.umn.edu/~shaden/ipdps17

Table I: Summary of datasets.

Dataset	NNZ	Dimensions	Size (GB)
Outpatient [21]	87M	1.6M, 6K, 13K, 6K, 1K, 192K	4.1
Netflix [22]	100M	480K, 18K, 2K	1.6
Delicious [24]	140M	532K, 17M, 3M	2.7
NELL [25]	143M	3M, 2M, 25M	2.4
Yahoo [23]	262M	1M, 624K, 133	4.3
Reddit [26]	924M	1.2M, 23K, 1.3M	15.0
Amazon [27]	1.7B	5M, 18M, 2M	36.4

NNZ is the number of nonzero entries in the dataset. **K**, **M**, and **B** stand for thousand, million, and billion, respectively. **Size** is the amount of memory in gigabytes required to represent the tensor in a single CSF.

B. Datasets

Table I details the tensors used in our evaluation. We selected tensors from a variety of real-world applications which extensively use the CPD. Outpatient is a six-mode *patient-institution-physician-diagnosis-procedure-date* tensor formed from synthetic electronic medical records [21]. Netflix [22] and Yahoo [23] are both *user-movie-date* tensors formed from movie ratings. Delicious is a *user-item-tag* tensor formed from user-supplied tags of websites [24]. NELL is a *noun-verb-noun* tensor from the Never Ending Language Learning project [25]. Reddit [26] is a *user-community-word* tensor representing a subset of user comments from Reddit[2] from 2007 to 2012. Amazon is a *user-item-word* tensor representing product reviews [27]. The Delicious, NELL, Reddit, and Amazon datasets are publicly available in the FROSTT collection [28].

C. Exploring Decompositions on Many-Core Processors

We first explore the performance implications of problem decomposition on a many-core processor. In order to separate the effects of decomposition and KNL-specific hardware features, we explicitly place all allocations in DDR4 memory and use one thread per core. We use a pool of 1024 OpenMP mutexes for synchronization.

1) Partial Tiling: Figure 4 shows the effects of tiling one, two, and three modes with a single CSF representation according to the strategy described in Section IV-A. No strategy consistently outperforms the others. Amazon sees the most benefit from tiling two and three modes, achieving a $4.5\times$ speedup over the untiled implementation. The large disparity between tiling one and multiple modes of Amazon is not due to synchronization costs, but due to load imbalance. We further explore this challenge in Section V-C4.

2) Privatization: The Netflix, Outpatient, Reddit, and Yahoo tensors have a combination of long and short modes. The short modes result in lock contention. Figure 5 shows the effects of privatization (Section IV-B) as we change the number of tiled modes with a single CSF representation. The two- and three-mode tiling schemes see small performance gains from privatization, but tiling a single mode achieves

[2]https://reddit.com/

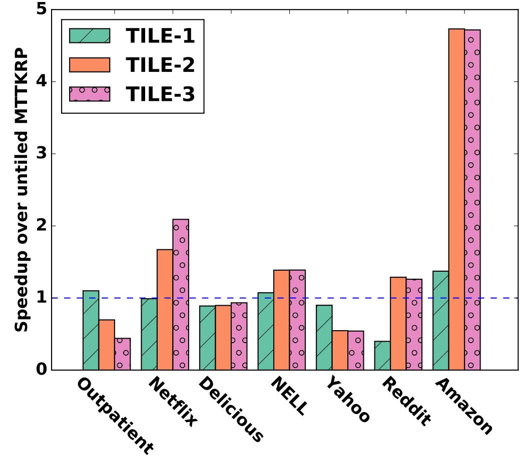


Figure 4: Speedup over untiled MTTKRP while tiling the longest (**Tile-1**), two longest (**Tile-2**), and three longest modes (**Tile-3**).

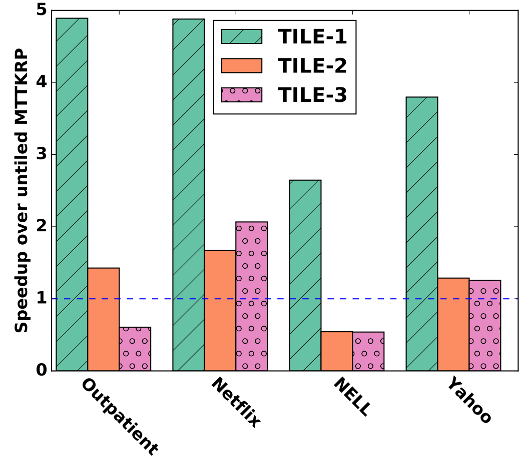


Figure 5: Speedup over untiled MTTKRP using one, two, and three tiled modes with privatization for synchronization. Privatized modes were selected per Equation 2 with $\gamma{=}0.2$.

significant speedups compared to untiled and also mutex-only synchronization (Figure 4). The slowdowns beyond a single tiled mode are attributed to the overheads of storing and operating with additional tiles. We use privatization with single-mode tiling for Netflix, Outpatient, Reddit, and Yahoo in the remaining experiments.

3) Number of CSF Representations: Figure 6 shows MTTKRP's performance as we increase the number of CSF representations. We follow the scheme presented in Section IV-A and use one, two, and M CSF representations. The tensors fall into two categories. Delicious and NELL benefit from a second and a third CSF representation, with diminishing returns after the second. The remaining tensors achieve the best performance with either one or two

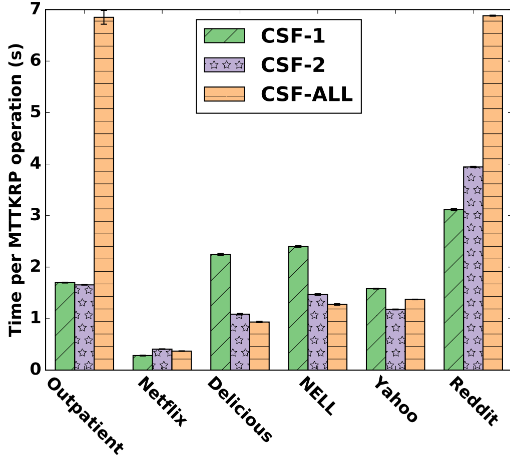


Figure 6: Effects of the number of CSF representations on MTTKRP runtime, using 1, 2, and M representations. Amazon is omitted due to memory constraints.

Table II: Load imbalance on the Amazon dataset.

	Load Imbalance				Time (s)			
	BDW		KNL		BDW		KNL	
Mode	**slice**	**hub**	**slice**	**hub**	**slice**	**hub**	**slice**	**hub**
1	0.72	**0.04**	0.84	0.05	2.37	0.71	3.22	**0.53**
2	0.13	0.04	0.05	**0.03**	1.31	0.79	0.73	**0.72**
3	0.07	**0.03**	0.24	0.18	2.67	2.61	1.96	**1.82**

Load imbalance is defined as the relative difference between the maximum and average time spent by all threads. **slice** denotes coarse-grained parallelism in which full slices are distributed to threads. **hub** denotes using fine-grained parallelism on "hub" slices and coarse-grained parallelism on all others.

representations. We note that these are the four tensors that have highly skewed mode lengths. When a short mode is moved from the top CSF level, the resulting tree structure is changed and often achieves less compression than before. Since we have already improved performance on the skewed tensors through partial tiling and privatization, there is little to be gained from additional representations.

4) Load Imbalance: Table II shows load imbalance and runtime on the Amazon dataset with one tiled mode. We measure load imbalance as the relative difference between the maximum and average time spent by all threads:

$$\text{imbalance} = \frac{t_{max} - t_{avg}}{t_{max}}.$$

The first mode of Amazon has a highly skewed distribution, with 6.5% of all non-zeros residing in a single slice. This overloaded slice prevents load balance for any coarse-grained (i.e., slice-based) parallelism.

BDW and KNL suffer from 72% and 84% load imbalance, respectively. When we switch to a fine-grained parallelism for the hub slices, load imbalance reduces to 4% and 5% on BDW and KNL, leading to 3.3× and 6.1× speedups, respectively. This resulting performance exceeds that of tiling with two and three modes.

Table III: Summary of the best known decompositions.

Dataset	Tiled Modes	CSF Reps.	Hub	Prv.
Outpatient	1	2		✓
Netflix	1	1		✓
Delicious	0	3	✓	
NELL	0	3	✓	
Yahoo	1	2		✓
Reddit	1	1		✓
Amazon	1	1	✓	

Tiled Modes is the number of tiled modes. **CSF** is the number of CSF representations. **Hub** indicates if there are any hub slices. **Prv.** indicates if we used privatization for at least one mode.

D. Harnessing the KNL Architecture

We now explore performance optimizations specific to the KNL architecture. Unless otherwise noted, we work from the best decompositions learned in the previous experiments, which are summarized in Table III. We note that every dataset in the evaluation benefits from at least one algorithmic contribution (i.e., partial tiling, privatization, multiple CSF representations, or hub slices).

1) MCDRAM: Figure 7 illustrates the benefits of MCDRAM over only DDR4 memory. We computed memory bandwidth by measuring the amount of data transferred from DDR4 or MCDRAM via hardware counters and divided this number by the time to compute MTTKRP [5].

Reddit and Amazon do not fit entirely inside of MCDRAM, and so we place only the factors inside of MCDRAM and measure the bandwidth from both memories. Interestingly, placing additional structures in MCDRAM (e.g., the tensor values or indices) does not improve performance due to KNL's ability to access DDR4 and MCDRAM concurrently. Any additional MCDRAM allocations simply increase the observed MCDRAM bandwidth while equally decreasing the observed DDR4 bandwidth, resulting in no net performance increase.

Outpatient is not memory-bound and sees little benefit from MCDRAM. We attribute this to its short mode lengths, which encourage temporal reuse. Additionally, Outpatient has a large number of modes, forcing it to incur high synchronization costs relative to the lower order tensors. We therefore omit it from the remaining MCDRAM evaluation.

When constrained to DDR4 memory, the remaining datasets are bounded by the maximum achievable bandwidth. MCDRAM increases the achieved bandwidth from 2.7× on Delicious to 3.7× on Netflix. The three datasets with the longest mode lengths (i.e., Delicious, NELL, and Amazon) are heavily dominated by read-bandwidth. NELL and Amazon achieve approximately 80% of the maximum 380 GB/s of read-bandwidth. The observed bandwidths do not fully saturate the MCDRAM's capabilities. We note that the MTTKRP time also includes computation which may not be overlapped with data movement, leading to an observed bandwidth which is lower than actually achieved. Thus, the presented bandwidth is a lower bound for the

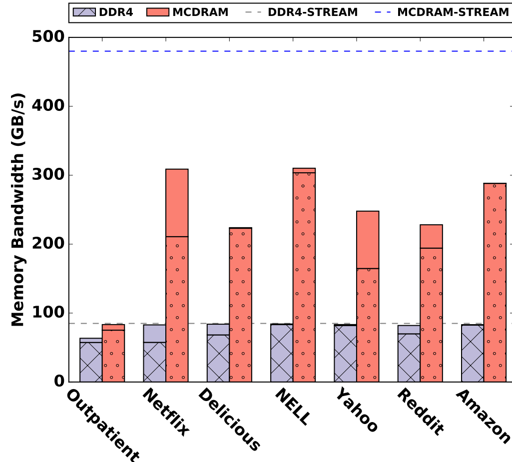


Figure 7: Observed memory bandwidth (BW) on KNL in flat mode with DDR4 and MCDRAM. Values indicate the maximum BW achieved for an MTTKRP kernel during CPD. Stacked bars encode read-BW (bottom) and write-BW (top). **DDR4-STREAM** and **MCDRAM-STREAM** indicate the maximum attainable read+write-BW per the STREAM benchmark [6]. KNL's maximum read-BW out of MCDRAM is 380 GB/s.

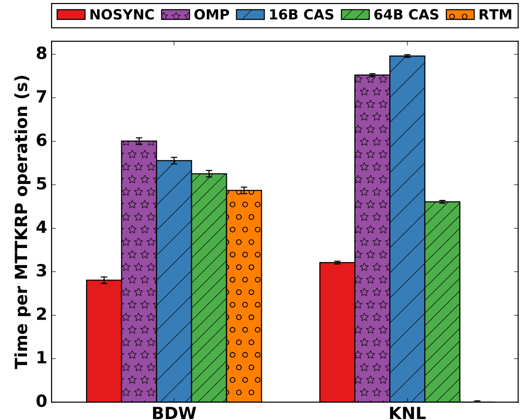


Figure 8: Comparison of synchronization primitives on the Outpatient dataset. All data is placed in MCDRAM on KNL.

achieved bandwidth. A more detailed profiling or a formal performance analysis would be beneficial in determining the precise achieved bandwidth and whether MTTKRP is still bandwidth-bound in the presence of MCDRAM. We leave these tasks to future work.

The two-level memory hierarchy provided by KNL facilitates large scale computations which could not be performed in the presence of only MCDRAM. Reddit and Amazon demonstrate that a problem does not need to entirely fit in MCDRAM to obtain significant speedup, and instead we can focus on the bandwidth-intensive data structures for MCDRAM allocation.

2) Synchronization Primitives: Figure 8 illustrates the overheads associated with various synchronization primitives during MTTKRP execution on KNL, compared to BDW. We report runtimes on Outpatient, which has the highest number of modes and also the shortest modes in our evaluation, and therefore the highest synchronization overheads. We do not use any tiling or privatization constructs in order to better evaluate the synchronization primitives provided by hardware. NOSYNC uses no synchronization and serves as a performance baseline. OMP uses a pool of 1024 OpenMP mutexes. 16B CAS uses 16B compare-and-swap (CAS) and 64B CAS simulates 64B CAS by issuing one 16B CAS for every 64B of data. RTM uses restricted transactional memory, which is available on BDW.

OMP and 16B CAS introduce over 100% overhead on KNL. The large vector instructions that AVX-512 offers are not well utilized with 16B CAS, as four CAS must be issued per fused multiply-add (FMA) instruction. The simulated 64B CAS reduces overhead to 30% due to KNL utilizing an entire AVX-512 FMA per CAS. Future many-core architectures could benefit significantly if CAS instructions are made wide enough to support the large vector registers.

RTM introduces the least overhead on BDW, including the simulated 64B CAS instructions. There is still a 42% overhead associated with RTM, however, suggesting that relying solely upon hardware-provided synchronization is insufficient for the best performance.

3) Simultaneous Multi-threading: KNL supports 4-way simultaneous multi-threading (SMT) as a method of hiding memory access latency. We examine the benefits of SMT in Figure 9. We execute in MCDRAM cache mode and run with 1, 2, and 4 threads per core for a total of 68, 136, and 272 threads. If a tensor is tiled, we fix the tile dimensions to be 272 and distribute additional tile layers to threads. Thus, each configuration performs the same amount of work and has the same sparsity structure. This allows us to eliminate the effects of varying decomposition while observing the effects of hiding latency.

Using two threads per core universally improves performance by masking access latencies. Performance is mixed beyond two threads due to increased synchronization costs resulting from lock contention or larger data reductions when using privatization. In the worst case, Outpatient spends $3.5\times$ more time on synchronization. We note that load imbalance is not affected due to the same decomposition being used across thread configurations. We recommend using two threads per core due to its consistent benefit.

E. Comparing BDW and KNL

Figure 10 shows the performance of best-performing decompositions on KNL and BDW. We include KNL in both cache and flat mode configurations.

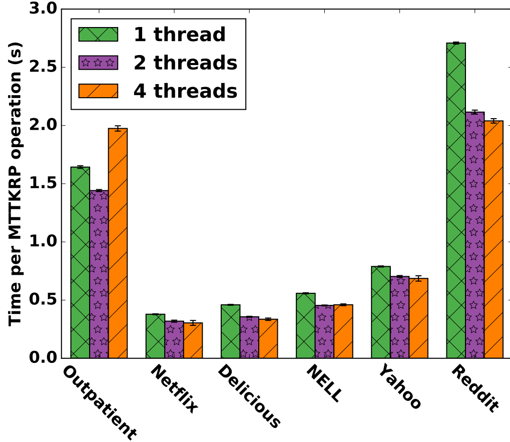


Figure 9: Effects of simultaneous multi-threading on KNL in cache mode. Amazon is omitted due to memory limitations when using four threads (due to the changed sparsity structure induced by tiling).

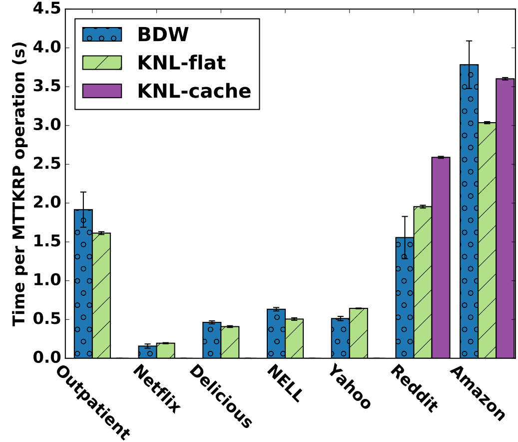
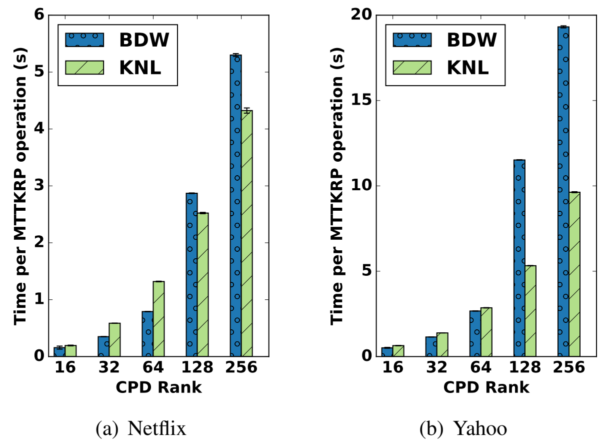


Figure 10: Comparison of MTTKRP performance on KNL and BDW. **KNL-flat** and **KNL-cache** denote KNL in flat and cache mode, respectively. Datasets which fit entirely in MCDRAM have identical running times in cache and flat mode and are thus omitted.

Observe that flat mode is up to 30% faster than cache mode when the dataset does not fit in MCDRAM. The tensor is accessed in a streaming fashion and exhibits no temporal locality, but still may be placed in the MCDRAM cache. By fixing the matrix factors in MCDRAM, which do exhibit temporal locality, we can ensure better utilization of the valuable MCDRAM resource.

KNL ranges from $0.84\times$ slowdown on Reddit to $1.24\times$ speedup on Amazon over BDW. Unsurprisingly, we can see that KNL is most advantageous on the large, sparse tensors which are bandwidth-bound and benefit the most from MCDRAM. The last-level cache (LLC) of BDW allows it to outperform KNL on tensors with short modes. Netflix, for example, only requires 64MB to store all three factors. We explore larger CPD ranks with Netflix and Yahoo in Figure 11. In both cases, BDW is either faster or competitive to KNL for the smaller ranks due to the small factors mostly fitting in BDW's large LLC. BDW sees a sharp slowdown between ranks 64 and 128 as the factors no longer fit in LLC. KNL then proceeds to outperform BDW up to $1.8\times$ due to MCDRAM's larger capacity.



(a) Netflix (b) Yahoo

Figure 11: Effects of increasing CPD rank on MTTKRP.

VI. Conclusions and Future Work

We presented the first exploration of sparse tensor factorization on a many-core processor, using the new Xeon Phi Knights Landing processor as a case study. We addressed challenges such as managing high-bandwidth memory, load balancing hundreds of threads, and reducing fine-grain synchronization. We showed that no parallelization or synchronization strategy works consistently across datasets and provided guidelines for deciding which strategies to employ that take into account various tensor properties. Our evaluation highlighted the need for improved hardware atomics on many-core architectures. Our algorithmic advancements achieved up to $4.9\times$ speedup over existing implementations. Lastly, we show that a Knights Landing processor can reduce time-to-solution up to $1.8\times$ compared to a 44-core Xeon system.

We presented several problem decompositions which improve MTTKRP performance. A limitation is the lack of *predictive* performance understanding – no decomposition is universally better than others. As future work, we will develop a performance model for MTTKRP which would allow one to better understand the performance benefits and limitations of each decomposition.

The computations presented in this work closely resemble those in other data intensive applications. We plan to explore the application of the presented optimizations to other data intensive problems on many-core architectures.

Acknowledgments

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), a University of Minnesota Doctoral Dissertation Fellowship, Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

References

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[2] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1265–1274.

[3] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowledge-Based Systems*, vol. 98, pp. 130–147, 2016.

[4] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver, "Tfmap: optimizing map for top-n context-aware recommendation," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2012, pp. 155–164.

[5] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[6] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[7] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *International Parallel & Distributed Processing Symposium (IPDPS'15)*, 2015.

[8] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2015, p. 7.

[9] A. Heinecke, A. Breuer, M. Bader, and P. Dubey, "High order seismic simulations on the intel xeon phi processor (knights landing)," in *International Conference on High Performance Computing*. Springer, 2016, pp. 343–362.

[10] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[11] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 273–282.

[12] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 339–350.

[13] M. Baskaran, B. Meister, and R. Lethin, "Low-overhead load-balanced scheduling for sparse tensor computations," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[14] N. Ravindran, N. D. Sidiropoulos, S. Smith, and G. Karypis, "Memory-efficient parallel computation of tensor and matrix products for big tensor decomposition," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2014.

[15] J. H. Choi and S. Vishwanathan, "DFacTo: Distributed factorization of tensors," in *Advances in Neural Information Processing Systems*, 2014, pp. 1296–1304.

[16] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 77.

[17] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS'16)*, 2016.

[18] T. B. Rolinger, T. A. Simon, and C. D. Krieger, "Performance evaluation of parallel sparse tensor decomposition implementations," in *Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE, 2016.

[19] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/~splatt/.

[20] H. Bae, J. Cownie, M. Klemm, and C. Terboven, "A user-guided locking api for the openmp* application program interface," in *International Workshop on OpenMP*. Springer, 2014, pp. 173–186.

[21] Center for Medicare and Medicaid Services. (2010) CMS data entrepreneurs synthetic public use file (DE-SynPUF). [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/Downloadable-Public-Use-Files/SynPUFs/index.html

[22] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.

[23] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11." in *KDD Cup*, 2012, pp. 8–18.

[24] O. Görlitz, S. Sizov, and S. Staab, "Pints: peer-to-peer infrastructure for tagging systems." in *IPTPS*, 2008, p. 19.

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

[26] J. Baumgartner. (2015) Reddit comment dataset. [Online]. Available: https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/

[27] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 2013, pp. 165–172.

[28] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/