SIAM J. SCI. COMPUT.
Vol. 41, No. 3, pp. C269–C290

SOFTWARE FOR SPARSE TENSOR DECOMPOSITION ON EMERGING COMPUTING ARCHITECTURES*

ERIC T. PHIPPS† AND TAMARA G. KOLDA‡

Abstract. In this paper, we develop software for decomposing sparse tensors that is portable to and performant on a variety of multicore, manycore, and GPU computing architectures. The result is a single code whose performance matches optimized architecture-specific implementations. The key to a portable approach is to determine multiple levels of parallelism that can be mapped in different ways to different architectures, and we explain how to do this for the matricized tensor times Khatri–Rao product (MTTKRP), which is the key kernel in canonical polyadic tensor decomposition. Our implementation leverages the Kokkos framework, which enables a single code to achieve high performance across multiple architectures that differ in how they approach fine-grained parallelism. We also introduce a new construct for portable thread-local arrays, which we call compile-time polymorphic arrays. Not only are the specifics of our approaches and implementation interesting for tuning tensor computations, but they also provide a roadmap for developing other portable high-performance codes. As a last step in optimizing performance, we modify the MTTKRP algorithm itself to do a permuted traversal of tensor nonzeros to reduce atomic-write contention. We test the performance of our implementation on 16- and 68-core Intel CPUs and the K80 and P100 NVIDIA GPUs, showing that we are competitive with state-of-the-art architecture-specific codes while having the advantage of being able to run on a variety of architectures.

Key words. tensor decomposition, canonical polyadic, MTTKRP, Kokkos, manycore, GPU

AMS subject classifications. 15A72, 15-04, 68W10, 68-04

DOI. 10.1137/18M1210691

1. Introduction.

Tensors, or multidimensional arrays, are a powerful means of representing relationships in multiway data [17]. We focus on computing the canonical polyadic or CANDECOMP/PARAFAC (CP) decomposition [11, 4] for sparse tensors. CP decompositions have numerous applications in data science, including analysis of online social networks [10], anomaly detection [9], compression of neural nets [13, 24, 6], and health data analytics [35], among others. The CP decomposition is a low-rank decomposition and approximates a given tensor by a sum of rank-one tensors. In this work, our focus is on the main computational kernel in computing the CP decomposition: the matricized tensor times Khatri–Rao product (MTTKRP). We describe the mathematical background on tensors and MTTKRP in section 3.

Our goal is to develop CP decomposition software that is *portable to and performant on* a variety of multicore and manycore computing architectures, such as multicore CPUs, the manycore Intel Xeon Phi, and NVIDIA GPUs. This means we desire software that runs on a variety of computing architectures, contains few architecture-specific optimizations, ports to new architectures with only small code modifications, and yet achieves performance on par with customized implementations

*Submitted to the journal's Software and High-Performance Computing section August 30, 2018; accepted for publication (in revised form) April 16, 2019; published electronically June 20, 2019.
http://www.siam.org/journals/sisc/41-3/M121069.html
Funding: This work was supported by the Laboratory Directed Research and Development (LDRD) Program at Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
†Sandia National Laboratories, Albuquerque, NM 87185 (etphipp@sandia.gov).
‡Sandia National Laboratories, Livermore, CA 94551-9159 (tgkolda@sandia.gov).

that have been optimized for each architecture. Relying on a single code implementation simplifies software development and maintenance while providing some degree of "future-proofing" as new architectures and programming models are developed.

The key to developing portable code is to identify multiple levels of parallelism that can be mapped in different ways to different architectures. For MTTKRP with a sparse tensor, we observe parallelism across the nonzeros and also across the columns of the factor matrices. To realize the portable parallelism in practice, we leverage the Kokkos package [7, 8], which provides threading abstractions that allow a single software implementation to be portable to a diversity of shared memory parallel programming models (such as OpenMP, pThreads, and CUDA) and hardware. For example, fine-grained parallelism on CPU and Xeon Phi architectures is expressed through vector arithmetic operations that are usually incorporated through automatic vectorization of low-level loops by the compiler. Conversely, fine-grained parallelism on GPUs is expressed through the explicit programming of groups of cooperating threads. Kokkos provides abstractions that unify these different approaches, making a *single* software implementation leveraging fine-grained parallelism feasible. The multiple levels of parallelism and use of Kokkos are described in section 4.

One issue with Kokkos is that it does not provide a good way to allocate thread-private arrays. To address this, we introduce a new portable and performant data abstraction based on compile-time polymorphic arrays in section 5, and we contrast this with Kokkos scratch pad arrays. Our method enables better vector parallelism at the lowest loop level, especially on GPUs, and, we contend, is simpler to use than what was already available in Kokkos. Using polymorphic arrays for fine-grained parallelism does not depend on Kokkos and could also be used in other contexts.

To avoid race conditions in thread-parallel writes in MTTKRP, we rely on atomic-write instructions since they are generally scalable for high concurrency architectures. Nevertheless, these atomic instructions can substantially reduce performance for many tensors, so a modification to the MTTKRP algorithm is described in section 6 that reorders operations to reduce contention among threads writing to the same memory location, with the tradeoff being storage of additional permutation arrays that doubles the size of the tensor stored in memory.

Our portable and performant software implementation of the CP alternating least squares (CP-ALS) method for sparse tensors is available in the open source GenTen library.[1] Section 7 provides performance results for this code with the different versions of MTTKRP discussed in the paper. We test on multicore CPUs, NVIDIA GPUs, and the Knights Landing (KNL) version of the Intel Xeon Phi. We also compare our code to the state-of-the-art open-source codes SPLATT [31], DFacTo [5], and ParTI! [20]. These results show that GenTen using the modified MTTKRP algorithm achieves state-of-the-art performance on a variety of platforms with a single codebase.

To summarize, this work provides three primary contributions. First and foremost, this work describes how to achieve performance portability for the MTTKRP kernel by identifying coarse and fine-grained parallelism and leveraging the Kokkos package to exploit it. Second, we introduce a novel construct that uses thread-local compile-time polymorphic arrays for leveraging fine-grained parallelism in MTTKRP. Third, we present a new permuted variant of MTTKRP that dramatically reduces costs associated with atomic-writes. Taken together, our publicly available GenTen software library incorporating these contributions represents the first-ever implementation of CP-ALS in a portable manner, achieving high performance on CPUs, Intel

[1] https://gitlab.com/tensors/genten

Xeon Phi, and NVIDIA GPUs with a single implementation.

2. Related work. SPLATT [31] stores each mode of a sparse tensor as a list of slices, where each slice is stored in a compressed format similar to the compressed sparse row/compressed row storage (CSR/CRS) format of sparse matrices. The MTTKRP algorithm is parallelized over the rows of the result matrix using a task parallelism scheme implemented through OpenMP. A cache blocking scheme is also introduced to improve MTTKRP performance. Because of the mode-dependent storage format, this approach requires a different representation for each mode of the tensor, which substantially increases memory costs. More recently, SPLATT incorporated a compressed sparse fiber (CSF) approach [29] that stores a tensor as a family of trees, which, in principle, avoids duplication of the tensor for each mode. However, because a mode must be chosen to form the roots of the trees, the resulting MTTKRP algorithm can have substantially different performance depending on which mode is being traversed. So, in practice, multiple CSF representations are computed and stored in memory to reduce total CP-ALS time. The MTTKRP algorithm incorporates thread parallelism and employs a tiling mechanism to avoid the use of locks or atomic instructions in order to handle thread race conditions. The performance of the CSF approach in SPLATT has also been recently explored on the KNL version of the Intel Xeon Phi architecture [30], where several variants of the MTTKRP algorithm were considered, including a thread privatization approach to handling race conditions. Conversely, our work considers the coordinate-based format for sparse tensor storage exclusively, which does not require any duplication of the tensor, and the use of atomic instructions for handling race conditions. Since the use of atomics on some architectures (such as KNL) can be quite slow, we also propose a modified MTTKRP algorithm that substantially reduces atomic contention, with a little less than twice the memory storage. We also consider portability to GPUs in our work.

Building on the CSF data structure, Li et al. proposed an adaptive tensor memoization algorithm that reduces the number of redundant floating-point operations that occur during the sequence of MTTKRP calculations required by CP-ALS, with the trade-off of increased memory usage [19] due to storing semisparse intermediate tensors. An adaptive model tuning framework called AdaTM was also developed that chooses an optimized memoization algorithm based on the sparse input tensor. Fine-grained parallelism across multiple modes of the intermediate tensors was proposed but only studied within the context of thread parallelism on multicore CPU architectures. Portability to KNL and GPU architectures is not considered.

DFacTo [5] stores a sparse tensor as a set of sparse matrices representing unfoldings of the tensor for each mode, and reorganizes MTTKRP for each mode as a sequence of sparse matrix-vector products (SpMV). As such DFacTo can rely on optimized implementations of SpMV, and therefore focuses on distributed memory parallelism. DFacTo doesn't consider portability explicitly, but reliance on an SpMV essentially provides it. However DFacTo's MTTKRP algorithm has only been developed for third-order tensors and requires storing a separate unfolding for each tensor mode. DFacTo stores these in a CSR format which makes the storage reasonably memory efficient, but requires packing all but one of the coordinates for each nonzero into a single ordinal, which may not fit if the tensor has long mode lengths. Furthermore, use of CSR-based SpMV for MTTKRP can be inefficient due to the usual row-based parallelization strategy if there is a wide spread in the number of nonzeros per tensor slice (resulting in imbalance in the number of nonzero columns in the unfolded tensor), or if the tensor has short mode lengths (resulting in insufficient par-

allelism). DFacTo was later rewritten as ReFacTo [27] to use vendor-optimized SpMV routines as well as run on Nvidia GPUs.

The Cyclops Tensor Framework [32] provides a general framework for implementing dense and sparse tensor operations upon which CP decompositions can be built. For sparse tensors, the library relies on matricization and a corresponding sparse matrix primitive, and therefore suffers the same limitations as DFacTo described above. The library is geared towards distributed memory parallelism.

Kaya and Uçar propose an MTTKRP algorithm based on dimension trees [15] to eliminate the redundant computations that appear when performing MTTKRP calculations for each mode during each iteration of CP-ALS. This approach views MTTKRP as a sequence of tensor-vector products, and stores intermediate products that appear multiple times in a sequence of MTTKRP calculations. This reduces computation by a factor of $(d-1)/\log d$, where d is the dimension of the tensor, with an increase in memory storage by at most a factor of $1+\log d$. The original tensor and intermediate tensors are stored in coordinate format. Both shared and distributed memory parallelism are considered, with the former parallelizing over the nonzero indices of each intermediate tensor (somewhat akin to parallelizing over rows of the resulting factor matrix as in the original SPLATT algorithm above). Hence race conditions are not an issue, but parallelism can be limited when there are short mode lengths. There work was encapsulated into the HyperTensor package using OpenMP and MPI parallelism. Portability to GPUs is not considered. While beyond the scope of this work, a performance portable implementation of dimension trees could be considered. The short mode-length issue would be particularly problematic on GPUs, but a solution involving the multilevel parallelism and portability techniques introduced here would likely be successful.

More recently, Liu et al. proposed an extension of coordinate-based tensor storage format called F-COO (flagged-coordinate) [22] which is extensible to many types of tensor operations, and demonstrated MTTKRP and CP decomposition results on GPUs. Portability to non-GPU architectures is not considered. F-COO modifies the traditional coordinate-based format (COO) by adding bit arrays that determine when a mode index changes and are used to implement a thread-parallel MTTKRP algorithm based on segmented reduction, thereby avoiding atomic instructions. The authors claim F-COO requires less memory storage than COO (since one index is replaced by bit arrays), but since MTTKRP operations are required for all modes in CP-ALS, storage of the bit flag arrays for all modes increases total memory storage by a small factor. The authors also claim that an MTTKRP algorithm based on atomics is too slow on the GPU, but as we will show in section 7, the algorithm performs very well on newer GPU models with fast atomic hardware. The F-COO approach is conceptually similar to the permutation-based approach we propose in section 6 which also reduces the costs associated with atomics. Instead of using bit arrays to implement segmented reduction, we instead change the order of traversal of nonzeros using permutation arrays to make segmented reduction unnecessary. Our approach also has the advantage of improving the locality of writes when updating the resulting factor matrix in MTTKRP, which while not important on GPUs, is quite important on cache-based architectures such as CPUs and the Xeon Phi. However, the F-COO approach does require somewhat less storage than the permutation approach we propose.

Finally, Li, Sun, and Vuduc proposed a variant of the coordinate format called HiCOO [21] that decomposes a sparse tensor into small sparse blocks, reducing the memory required to store tensor nonzeros (and hence memory bandwidth to read

them). A thread-parallel MTTKRP algorithm is developed for small thread-count architectures such as CPUs that groups the small blocks into larger blocks called superblocks with the collection of superblocks distributed across threads. Race conditions are handled either by parallelizing between superblocks across the mode that is being written to when the length of that mode is large, or by using a thread privatization strategy when the mode length is small. Portability to GPUs is not considered, and it is unclear how the algorithm would be parallelized on GPUs with very large numbers of threads. Likely, the best approach would be to forego the two-level blocking strategy and instead parallelize directly between the small tensor blocks using atomics for handling race conditions (at least for recent GPUs with fast atomics). In this case, many of the portability ideas introduced in our work could be applied to this approach.

3. CP tensor decomposition and MTTKRP. Except for specific relevant concepts discussed in detail later, we assume in this section a basic familiarity with tensors and refer the reader to Kolda and Bader [17] for further details. Let $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ be a given d-way tensor. We refer to each way or dimension as a *mode*. For a given rank R, the goal is to find a low-rank model tensor $\mathcal{M}$ that is a good approximation to $\mathcal{X}$, i.e.,

$$\min_{\mathcal{M}} \ \|\mathcal{X} - \mathcal{M}\| \quad \text{s.t.} \quad \mathcal{M} = \sum_{j=1}^{R} \lambda_j \, \mathbf{a}_j^{(1)} \circ \mathbf{a}_j^{(2)} \circ \cdots \circ \mathbf{a}_j^{(d)}, \tag{3.1}$$

where λ_j is a scalar weight, $\mathbf{a}_j^{(n)}$ is a column vector of size I_n that is assumed to be normalized to unit-norm in some norm, and $\circ$ represents the tensor outer product. For notational convenience, we assemble all the column vectors in mode n into a *factor matrix* of size $I_n \times R$:

$$\mathbf{A}^{(n)} = \begin{bmatrix} \mathbf{a}_1^{(n)} & \cdots & \mathbf{a}_R^{(n)} \end{bmatrix}.$$

Hence, the goal is to find the *weight vector* $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_1 & \cdots & \lambda_R \end{bmatrix}^{\mathsf{T}}$ and the d factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(d)}\}$ that define the low-rank model tensor $\mathcal{M}$. Following [2], we refer to $\mathcal{M}$ as a Kruskal tensor, or K-tensor for short.

In this work, we compute the CP decomposition using the CP-ALS method [11, 4]. Details are omitted here but can be found in, e.g., the survey by Kolda and Bader [17]. Our main interest is in the MTTKRP calculation for a sparse tensor, so we focus on this kernel for sparse tensors for the remainder of this section.

We say a tensor is *sparse* if the majority of its elements are zero. We can store such a tensor efficiently by storing only its nonzeros and their indices [1]. Here, we denote the ith nonzero and its subscripts as x_i and $(\ell_{i1}, \ell_{i2}, \ldots, \ell_{id})$, respectively. If there are P nonzeros, then we store $\mathcal{X}$ with a P-vector of real values and a $P \times d$ vector of coordinates. As discussed in section 2, a variety of sparse tensor formats have been proposed in the literature [2, 31, 5, 22, 21]. Storing the tensor $\mathcal{X}$ as list nonzero indices and values [2] is called *coordinate format* (COO), and this is what we use here since it does not favor any particular tensor mode.

For a sparse tensor in COO format, the mode-n MTTKRP computes a matrix $\mathbf{V}$ of size $I_n \times R$ that is defined elementwise as [2]

$$v(k,j) = \lambda_j \sum_{\substack{i=1 \\ \ell_{in}=k}}^{P} x_i \prod_{\substack{m=1 \\ m \neq n}}^{d} a^{(m)}(\ell_{im}, j) \quad \text{for} \quad k = 1, \ldots, I_n \text{ and } j = 1, \ldots, R. \tag{3.2}$$

TABLE 1
Levels of parallelism.

Level	CPU interpretation	GPU interpretation
League	Collection of teams	Grid (group of thread blocks), cannot synchronize
Team	Collection of hyperthreads on one or more cores	Thread block (usually 128 to 256 threads), shares fast memory and can synchronize
Vector	Vector instructions on single thread	Threads within GPU warp (i.e., 32 threads), execute in SIMD fashion

We assume d is small, ranging from 3–5 in the examples we show. In order of magnitude, P ranges from 10^6–10^8, I_n usually ranges from 10^3–10^6 but can be as small as 2, and R usually ranges from 10–100. The MTTKRP is the primary bottleneck of CP-ALS and the primary focus of our parallelization efforts.

4. Parallelism in MTTKRP. In this section we describe the multilevel parallelism present in MTTKRP and a portable implementation. Our goal is to be able to efficiently exploit fine-grained parallelism, by which we refer to the Single Instruction Multiple Data (SIMD) parallelism provided by vector instructions on multicore CPUs and the manycore Intel Xeon Phi, as well as the Single Instruction Multiple Thread (SIMT) parallelism provided by the fine-grained threads within a warp on CUDA (NVIDIA) GPUs. Ideally, we can use a single code that runs on any architecture, which is referred to as *portable*. Our portable implementation uses Kokkos [7, 8], a programming model and C++ library that enables applications and domain libraries to implement thread scalable algorithms that are efficient and portable across modern architectures. To discuss the levels of parallelism in a way that is agnostic to specific architectures, we use terminology mirroring the nested parallelism concepts introduced in the OpenMP 4.0 specification [25] consisting of a league of teams, where each team is comprised of a collection of threads, and each thread may execute vector instructions in parallel. The league is virtual (i.e., not tied to any hardware resource) and corresponds to the highest level parallel iteration space, while a team within the league corresponds to a collection of one or more hyperthreads on a CPU/Phi architecture or a thread block on a GPU architecture, and vector parallelism corresponds to CPU/Phi vector instructions or threads within a GPU warp. See Table 1 for a summary.

4.1. Multilevel parallelism for MTTKRP. Our goal is to exploit as much available parallelism in the MTTKRP calculation as possible. We use a multilevel parallelism approach, incorporating parallelism over tensor nonzeros, i.e., the i index in (3.2), and factor matrix columns, i.e., the j index in (3.2). For the latter we use vector-level parallelism, since it provides the best performance when iterating over contiguous regions in memory to enable packed/coalesced memory accesses,[2] which is possible with the combination of a rowwise layout of the factor matrices and a row-oriented MTTKRP algorithm that processes (portions of) factor matrix rows simultaneously. A pseudo-code description of the algorithm is shown in Figure 1. The variables should generally be the same as in (3.2), but the less intuitive mappings between the source code variables and the math are given in Table 2. Several param-

[2]On caching architectures such as CPUs and the Intel Xeon Phi, packed accesses refers to a given thread accessing consecutive memory locations sequentially and is a prerequisite for transforming a loop to use vector instructions. Conversely, coalesced accesses on a GPU refers to consecutive threads accessing consecutive memory locations, which is required to achieve full memory bandwidth.

```
// Compute MTTKRP with sparse tensor X, using K-Tensor M, in mode n.
// The result is stored in the factor matrix V.
mttkrp(Sptensor X, Ktensor M, unsigned n, FacMatrix V) {

  // Problem parameters
  P = X.nnz()               // Number of nonzeros
  d = M.ndims()             // Number of dimensions
  R = M.ncomponents()       // Number of factor matrix components

  // Architecture-specific values
  vector_size = ...             // Vector size (see Table 3)
  FBS = ...                     // Factor matrix column block size (see Table 3)
  team_size = ...               // Team size (see Table 3)
  NZPTM = 128                   // Nonzeros per team member
  NZPT = NZPTM * team_size      // Nonzeros per team
  league_size = (P+NZPT-1)/NZPT // Number of tensor nonzero blocks (league size)

  // Top level of parallelism...
  parallel_league_for(league_rank=0; league_rank<league_size; ++league_rank) {
    tmp = ScatchPad(team_size, FBS) // shared within the team

    // Loop over factor matrix column blocks
    for (jb=0; jb<R; jb+=FBS) {
      nj = (jb+FBS < R) ? FBS : R-jb  // number of columns to process

      // Second level of parallelism...
      parallel_team_for(team_rank=0; team_rank<team_size; ++team_rank) {
        i_offset = league_rank*NZPT + team_rank*NZPTM  // starting nonzero index
        ni = (i_offset+NZPTM < P) ? NZPTM : P-i_offset // number of nonzeros

        // Loop over tensor nonzeros in block
        for (i=i_offset; i<i_offset+ni; ++i) {
          x_val = X.value(i)     // value for nonzero i
          k = X.subscript(i,n)   // mode-n index for nonzero i

          // Initialize to x-value times lambda-weight
          parallel_vector_for(j=0; j<nj; ++j)
            tmp(team_rank,j) = x_val * M.weights(jb+j)

          // Multiply by corresponding factor matrix entries
          for (m=0; m<d; ++m)
            if (m != n)
              parallel_vector_for(j=0; j<nj; ++j)
                tmp(team_rank,j) *= M[m].entry(X.subscript(i,m),jb+j)

          // Multiple teams may be contributing to the same entries of the result
          parallel_vector_for(j=0; j<nj; ++j)
            atomic_add(V.entry(k,jb+j), tmp(team_rank,j))

        } // i
      } // team_rank
    } // jb
  } // league_rank
} // function
```

FIG. 1. MTTKRP-A. *Pseudo-code description of efficient parallel MTTKRP calculation in a C++-like syntax. Several of the algorithmic parameter choices are architecture dependent; see Table 3. It has three levels of parallelism at the league, team, and vector levels.*

eters are architecture dependent, and the values for those under different situations are shown in Table 3.

The factor matrices and subscripts are stored in row-major order, regardless of the architecture. The highest-level parallelism is indicated in the pseudo-code by the `parallel_league_for` loop. Each team within the league processes a mega-block of nonzeros of total size `NZPT`. The medium-level parallelism is indicated by the `parallel_team_for` loop. Each team member processes a nonzero block of size `NZPTM`, which we set to 128 for all experiments. The fine-level parallelism is indicated

TABLE 2
Math-to-code translations.

Code per Figure 1	Math per (3.2)
`X.value(i)`	x_i
`X.subscript(i,n)`	ℓ_{in}
`M.weights(j)`	λ_j
`M[m].entry(X.subscripts(i,m),j)`	$a^{(m)}(\ell_{im}, j)$
`V.entry(k,j)`	$v(k,j)$

TABLE 3
Choices for architecture-specific parameters in Figure 1.

Parameter	CPU	GPU
`FBS`	$\min\{\, 2^{\lceil \log_2(R) \rceil}, 32 \,\}$	
`vector_size`	1	$\min\{\,$ `FBS`$, 16\,\}$
`team_size`	1	128 / `vector_size`

by the `parallel_vector_for` loop. The vector parallelism is over a range of factor matrix columns, as each nonzero is processed.

The vector-level of parallelism over column indices exploits data locality in the factor matrices and subscripts. However, this requires allocation of a temporary buffer `tmp`. For efficiency, this buffer needs to be allocated in a fast memory space (such as CPU/Phi cache or GPU shared memory), which is limited in size. Hence, we work in blocks of factor matrix columns since we assume it is not possible to store an entire factor matrix row at one time when R is large. The factor matrix column loop is blocked by a run-time–determined tile size, `FBS`, where each `parallel_vector_for` only processes `FBS` columns at one time. Except when R is small, `FBS` $= 32$ (see Table 3), ensuring fully coalesced factor matrix accesses on GPUs and ample opportunity for vectorization on CPU/Phi. Although the tensor coordinates are reread at each iteration of the factor matrix block loop, the tensor nonzero block size (`NZPTM`) is small enough that these values fit in the cache across factor matrix block iterations. Because multiple nonzeros across multiple teams may contribute to the same entry of $\mathbf{V}$, we use `atomic_add` to resolve race conditions in writes to $\mathbf{V}$.

4.2. Implementation using Kokkos. We use Kokkos to implement Figure 1 in a manner that achieves high performance and portability. We refer the reader to the GenTen source code[3] to see the full implementation.

Kokkos provides a data structure called `View` for storing multidimensional arrays of data with template parameters specifying the type of data, its number of dimensions, the memory space in which data is allocated, and its layout in memory. Using `View`, we store the indices of the sparse tensor as a two-dimensional array of ordinals and the nonzero values as a one-dimensional array of floating-point data. The ordinal and floating-point types can be chosen while configuring GenTen, and we use `size_t` and `double`, respectively, in all numerical results below. The total size of a tensor $\mathcal{X}$ in memory is then $(ds_o + s_f) \cdot \text{nnz}(\mathcal{X})$ bytes, where s_o and s_f are the sizes in bytes of the ordinal and floating-point types, respectively, and $\text{nnz}(\mathcal{X})$ is the number of nonzeros in $\mathcal{X}$. We also use `View` to store each factor matrix as a two-dimensional array, and we store both tensor coordinates and factor matrices using Kokkos' rowwise memory

[3] https://gitlab.com/tensors/genten

layout to ensure packed/coalesced accesses of this data in the row-oriented MTTKRP algorithm discussed above.

Kokkos provides functions for the `parallel_for` commands in Figure 1 that are specialized for each architecture according to their corresponding parallel programming libraries (e.g., OpenMP or CUDA). In particular, the function corresponding to the `parallel_vector_for` command on a CPU/Phi maps to standard `for` loop that is intended to be autovectorized by the compiler. Conversely on a GPU, the loop iterations are mapped to individual threads within a warp according to their thread index. They each take their code bodies in the form of lambda-expressions, and accept a policy argument that describes the parallel iteration space. Kokkos provides mechanisms for allocating and managing the per-team temporary buffer `tmp`, which will be allocated in shared memory on a GPU and a fast cache on CPU/Phi. For `atomic_add`, Kokkos calls architecture-specific built-in atomic instructions when possible.[4]

5. MTTKRP with compile-time polymorphic arrays. Using Kokkos for the implementation of Figure 1 is portable and provides good performance on most architectures. It makes effective use of fine-grained vector parallelism, particularly when R is a multiple of the vector width of the architecture, and provides high memory-bandwidth efficiency. Even so, it does have drawbacks compared to optimized implementations for GPU and CPU architectures since it requires allocating the temporary buffer `tmp`. On a GPU, the buffer is stored in shared memory, which is very fast compared to global memory. However, the amount of shared memory available is quite small (typically 48–96 KB and is shared by all active threads on the GPU processor) requiring the factor matrix tile size (see Table 3) to be small so that enough thread blocks can be inflight to better hide the latency of global memory accesses. We would instead prefer to store the temporary values in registers, which are even faster to access and provide even more storage space (modern GPUs typically provide 256 KB for registers). Such a modification would allow the tile size to be larger and therefore improve performance. Similarly, on a CPU/Phi architecture, it would be more convenient to allocate the buffer as a simpler thread-private stack array.

To address these shortcomings, we propose a better-performing and arguably simpler approach to portability, which we refer to as compile-time polymorphic arrays. We extend Kokkos by introducing the class `TinyVec` that has different implementations depending on the architecture, as shown in Figure 2, which mirror the optimizations described above. The current scratch-pad capability of Kokkos creates a temporary buffer that is visible to all threads within a team; in contrast, we store the data in a thread-private manner. The wrapper class has several template parameters, including the execution space (allowing unique implementation for each architecture through partial template specialization), the length of the array (which therefore must be a compile-time constant), a flag indicating whether all or just a portion of the array will be used, and the chosen vector size (which also must be a compile-time constant). This class avoids the use of lambda expressions at the vector level, making it easier for the compiler to optimize and vectorize those loops.

On a CPU/Phi architecture (see Figure 2a), the array will be allocated as a

[4]The availability of those instructions depends upon the hardware and the data type. For example, NVIDIA K80 GPUs have hardware atomic instructions for single-precision data but not double-precision. If hardware instructions are not available, Kokkos uses a general implementation via atomic compare-and-swap (CAS), which is dramatically slower if there is high thread contention.

```
template <typename Space, unsigned Length, unsigned Size, unsigned VectorSize,
          typename Enabled = void>
struct TinyVec {
  integral_nonzero_constant<unsigned,Size> sz;
  alignas(64) double v[ Length ];

  TinyVec(const unsigned size, const double x) : sz(size) {
    for (unsigned i=0; i<sz.value; ++i) v[i] = x;
  }

  void store_plus(double* x) const {
    for (unsigned i=0; i<sz.value; ++i) x[i] += v[i];
  }

  TinyVec& operator+=(const TinyVec& x) {
    for (unsigned i=0; i<sz.value; ++i) v[i] += x.v[i];
    return *this;
  }

  void atomic_store_plus(volatile double* x) const;
  TinyVec& operator=(const double x);
  TinyVec& operator*=(const double x);
  TinyVec& operator*=(const double* x);
};
```

(a) Non-GPU architectures.

```
template <unsigned Length, unsigned Size, unsigned VectorSize>
struct TinyVec< Kokkos::Cuda,Length,Size,VectorSize,
                typename std::enable_if<Length/VectorSize == 4>::type > {
  integral_nonzero_constant<unsigned_type,Size/VectorSize> sz;
  double v0, v1, v2, v3;

  __device__ inline TinyVec(const unsigned size, const double x)
    : sz( (size+VectorSize-1-threadIdx.x) / VectorSize ) {
    v0 = v1 = v2 = v3 = x;
  }

  __device__ inline void store_plus(double* x) const {
    if (sz.value > 0) x[               threadIdx.x] += v0;
    if (sz.value > 1) x[  VectorSize + threadIdx.x] += v1;
    if (sz.value > 2) x[2*VectorSize + threadIdx.x] += v2;
    if (sz.value > 3) x[3*VectorSize + threadIdx.x] += v3;
  }

  __device__ inline TinyVec& operator+=(const TinyVec& x) {
    v0 += x.v0; v1 += x.v1; v2 += x.v2; v3 += x.v3;
    return *this;
  }

  __device__ inline void atomic_store_plus(volatile double* x) const;
  __device__ inline TinyVec& operator=(const double x);
  __device__ inline TinyVec& operator*=(const double x);
  __device__ inline TinyVec& operator*=(const double* x);

};
```

(b) GPU/CUDA architecture when `Length/VectorSize == 4`.

FIG. 2. *Example implementation of thread-local compile-time polymorphic array wrapper class* (`TinyVec`) *for two architectures. The implementations of several functions, which are similar to the implementations shown, are suppressed for brevity. The portion of the array used within the class is stored within* `integral_nonzero_constant<unsigned,Size>::value` *which will be* `Size` *when* `Size != 0`*, and its constructor argument otherwise.*

thread-private stack array of the given length (the factor matrix block size `FBS`). For all but the last iteration of the factor matrix block loop, the size flag indicates the *compile-time* full array length will be used. For the last iteration, the flag changes to indicate a *run-time–determined* size will be used to handle the remainder term when R is not evenly divisible by `FBS`.

Conversely on a GPU architecture (see Figure 2b), the array data is divided up among all of the threads in a warp (determined by the vector size). For example, if the array length is 128 and the vector size is 32, each thread holds four entries of the array. This data could be stored in a stack array as well, which will be converted to registers by the compiler when the full array length is used. When a dynamically sized portion of the array is used, the compiler instead stores this data in "local" memory which has the same high latency for accesses as global memory. To remedy this, we directly store the data in registers by providing *partial specializations* of the class based on the number of array elements per thread, and currently specializations are provided for 1–4 elements per thread. The specialization uses typical class template specialization techniques in conjunction with Substitution Failure Is Not An Error (SFINAE) [33] using `std::enable_if` for the `Enabled` template parameter to make the specialization available only when `Length/VectorSize == 4`. We refer the reader to the source code for complete details.

In all cases, the array wrapper provides overloaded operators for arithmetic on the array entries, where the implementation of each operator maps the operation across the entries of the array. On a CPU/Phi architecture, these can be well-optimized and vectorized by the compiler since they are simple, lowest-level loops, and when the full array is used, have a compile-time known trip count. On a GPU architecture, the operation is merely applied to each register variable. Thus, each vector-parallel loop over factor matrix entries that would have been implemented through a lambda-expression is replaced by a single overloaded operator call implemented by the array wrapper. This makes the code simpler and more understandable. Furthermore, since shared memory is no longer used to store the temporary buffer, a larger factor matrix tile size can be used, increasing performance. The resulting MTTKRP algorithm is displayed in Figure 3.

Conceptually, our array wrapper is similar to SIMD data types studied by others [18, 14, 34, 16, 26], which have been used to achieve some form of outer-loop vectorization by blocking the outer loop based on the architecture's vector width and moving the vector loop to an innermost loop encapsulated by overloaded operators iterating over a statically-sized array. Furthermore, these implementations have focused exclusively on vector CPU/Phi architectures with compile-time fixed array lengths. The contribution here is primarily the idea of using this technique for applying fine-grained parallelism across innermost loops, for GPU and CPU/Phi architectures. This requires a polymorphic API for reading and writing data to memory as well as the ability to support run-time choice of the array length for tail loops. We overcome performance issues peculiar to GPU architectures by storing the array entries in registers.

To achieve good performance for a wide range of R values, GenTen must determine choices of the factor matrix block size and vector size. Our logic for doing so tries to balance large factor matrix tile sizes (which reduce rereads of the tensor) with small remainders for the factor matrix loop (since the remainder portion is inherently less efficient). The result of our logic is displayed in Table 4, which shows the factor matrix block and vector sizes for ranges of R values (the vector size is only relevant for the GPU architecture; it is always one on a non-GPU architecture). On the GPU, these

```
parallel_league_for(league_rank=0; league_rank<league_size; ++league_rank) {

  // No shared team scratchpad!

  for (jb=0; jb<R; jb+=FBS) {
    nj = jb+FBS < R ? FBS : R-jb

    parallel_team_for(team_rank=0; team_rank<team_size; ++team_rank) {
      i_offset = league_rank*NZPT + team_rank*NZPTM
      ni = i_offset + NZPTM < P ? NZPTM : P-i_offset

      for (i=i_offset; i<i_offset+ni; ++i) {
        x_val = X.value(i)
        k = X.subscript(i,n)

        // Allocate special thread-local buffer
        tmp = TinyVec(nj,x_val) // Vector op
        tmp *= &M.weights(jb) // Vector op

        for (m=0; m<d; ++m)
          if (m != n)
            tmp *= &M[m].entry(X.subscript(i,m),jb) // Vector op

        tmp.atomic_store_add(&V.entry(k,jb)) // Vector op

      } //i
    } // team_rank
  } // jb
} // league_rank
```

FIG. 3. MTTKRP-B. *Modification of MTTKRP-A (Figure 1) using* `TinyVec` *polymorphic arrays (all code not shown is unchanged). In this case, we use different choices for the architecture-dependent parameters* `vector_size` *and* `FBS` *as shown in Table 4. The template parameters for* `TinyVec` *have been suppressed, including the logic for how the template parameters are determined to handle the remainder term. The vector operations span the range* `jb` *to* `jn+nj-1` *in the objects being accessed/written.*

TABLE 4

GPU vector and factor block sizes for different numbers of components (R) for the polymorphic array MTTKRP algorithm.

Range for R	1	2	3	4	5–7	8	9–16	17–24	25–47	48	49–95	96	97–
`vector_size`	1	2	2	4	4	8	8	8	8	16	16	32	32
`FBS`	1	2	4	4	8	8	16	24	32	48	64	96	128

choices result in at most four factor matrix columns processed by each thread within a warp. Notice that because of the greater amount of memory available for registers as opposed to shared memory, a much larger factor matrix block size is possible, compared to the scratch-pad based approach in Figure 1.

6. Permutation approach for MTTKRP. Depending on how the nonzeros are ordered in the tensor, many threads may be trying to update partial contributions to the result factor matrix simultaneously, creating high contention for the architecture's atomic hardware. This is particularly pronounced when there is no atomic-add instruction, such as `double` on NVIDIA Kepler architectures (e.g., K20, K40, K80), where Kokkos resorts to atomic CAS.

One approach for overcoming this challenge is to use a compressed storage format akin to the matrix compressed row storage (CRS) format, as is done in SPLATT [31]. For a given mode n, the tensor is stored as a list of slices for each element of that mode. The MTTKRP algorithm is parallelized over these slices where each thread operates on a distinct row of the resulting factor matrix, so no atomic update is necessary.

The downside of this approach for CP-ALS is that a separate copy of the tensor is required for each mode, since CP-ALS requires MTTKRP calculations for all modes.

In this work we introduce a new way of reducing atomic contention that is inspired by GPU implementations of the sparse matrix-vector product with sparse matrices stored in the COO format [3]. In this algorithm, the sparse matrix-vector product is parallelized over the matrix nonzeros. However, instead of having each thread write its contribution using an atomic instruction, the matrix nonzeros are sorted with increasing row index. Each thread iterates over a contiguous set of matrix nonzeros, and writes only occur when the row index changes. When a write is necessary, all the threads within a team perform a thread-parallel segmented reduction based on the row index, combining the contributions across multiple threads without atomics. Depending on how the algorithm is implemented, no atomic instructions may be necessary at all.

This approach cannot be directly applied to the tensor case because we want to avoid requiring d copies of the tensor, each with a different sorting. Instead we compute a permutation array for each mode that sorts the tensor nonzeros in increasing index along that mode. For mode n MTTKRP, we iterate over the tensor nonzeros in that permuted order as opposed to the order in which they are stored in memory. For a given tensor with P nonzeros and d modes, storing the tensor in memory requires $S = (s_r + ds_o)P$ bytes, where s_r and s_o are the sizes of the floating-point and ordinal types, respectively. This approach requires storing d additional permutation arrays of length P resulting in a total storage size of $(s_r + 2ds_o)P < 2S$ bytes, and therefore a little less than double the amount of memory is required to store the tensor. The downside of this approach is that the tensor nonzeros are no longer streamed from memory and are accessed in a more random fashion. However, each tensor nonzero corresponds to $O(d \cdot F)$ floating-point operations where F is the factor matrix tile size, allowing for significant reuse of those values.

This idea requires only small modifications of the MTTKRP kernel implementation so that each tensor nonzero index is extracted from the permutation array. The same logic is used for determining the vector and factor matrix tile sizes as shown in Table 4. Each thread within a team iterates over a given block size of tensor nonzeros and writes its contribution to the resulting factor matrix only when the mode-n coordinate changes. This must be an atomic-write if the mode-n index is equal to the first or last index of the block (since another thread may be writing to the same row); otherwise, it is a regular (nonatomic) write. Determining when and what kind of write should happen results in most of the changes to the kernel implementation, as shown in Figure 4. We could reduce contributions across threads within a team as is typically done in the sparse matrix-vector product algorithm to reduce the frequency of atomic-writes even further; however, this requires synchronization within each team, which appears to offset any potentially improved performance induced by the inter-team reduction.

There is a small preprocessing cost associated with this method to compute the permutation array for each tensor mode. GenTen can compute the permutation array using parallel sorting routines provided by Kokkos, as well as Thrust[5] for GPU architectures and the Intel Parallel Stable Sort[6] for OpenMP-based architectures.

[5] http://docs.nvidia.com/cuda/thrust/index.html

[6] https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp

```
parallel_team_for(team_rank=0; team_rank<team_size; ++team_rank) {
  i_offset = league_rank*NZPT + team_rank*NZPTM
  ni = i_offset + nzptm < P ? nzptm : P-i_offset
  val = TinyVec(nj,0.0) // accumulation buffer for a row, initialize to zero
  row_prev, first_row = -1 // used to track when to store

  // Loop over tensor nonzeros in block, in order wrt mode n
  for (i=i_offset; i<i_offset+ni; ++i) {
    p = X.getPerm(i,n)      // index of ith nonzero in permuted ordering
    x_val = X.value(p)      // value for nonzero p
    row = X.subscript(p,n) // mode-n index for nonzero p

    // Is this the first row in a block?
    if (i == i_offset) {
      first_row = row
      row_prev = row
    }

    // Detect change in row index
    if (row != row_prev) {
      // Sum the result into the appropriate entries in V
      if (row_prev == first_row) // First row needs atomic operation
        val.atomic_store_plus(&V.entry(row_prev,jb))
      else // Row owned entirely by this process, no atomic operation
        val.store_plus(&V.entry(row_prev,jb))

      //Reset for next row
      val = 0
      row_prev = row
    }

    tmp = TinyVec(nj, x_val)
    tmp *= &M.weights(jb)

    for (unsigned m=0; m<d; ++m)
      if (m != n)
        tmp *= &(M[m].entry(X.subscript(p,m),jb))

    // Accumulate sum across nonzeros until row index changes
    val += tmp

    // Last row needs atomic operation
    if (i == offset+ni-1)
      val.atomic_store_plus(&V.entry(row,jb)) // Vector operation

  } // i
} // team_rank
```

FIG. 4. MTTKRP-C. *Modification of MTTKRP-B (Figure 3) to use permuted indexing and avoid most atomic operations (all code not shown is unchanged).*

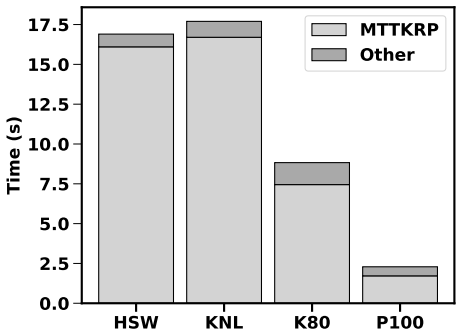
7. Numerical results. We investigate the performance of the different proposed portable parallel implementations of CP-ALS. Our results are generated with the publicly available GenTen library on four different architectures (both CPU and GPU). The specific architectures and compilers are listed in Table 5, with the architecture- and compiler-specific optimization flags specified by Kokkos.

7.1. Artificial data and scalability studies. We first consider running the full CP-ALS method on a synthetic three-dimensional tensor of size $30,000 \times 40,000 \times 50,000$ with ten million nonzeros placed randomly throughout the tensor. For each architecture, Figure 5 displays the total run-time for ten iterations of CP-ALS with $R = 128$ factor components using MTTKRP-B (Figure 3). Figure 5a displays the total run-time for all four architectures at full machine capacity. First, MTTKRP takes most of the CP-ALS computation time, regardless of architecture. Second, the Pascal P100 GPU is substantially faster than the other three. As compared to

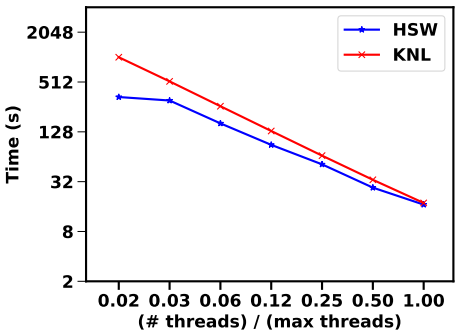
TABLE 5

Computational parallel architectures used for experimental results. While the KNL architecture supports up to 272 threads, at most 256 threads (64 cores) were used for our experiments. The KNL was placed in cache mode, where the high-bandwidth memory (HBM) is used as a last-level cache for main memory.

Type	Name	Architecture description	Thread parallelism	Compiler
CPU	Haswell	Intel Xeon E5-2698v3 CPU, 2.3 GHz, 2 sockets, 16 cores/socket, 2 threads/core, max 32 threads	OpenMP	Intel 17.0
CPU	KNL	Intel Xeon Phi 7250, 68 cores, 4 threads/core, HBM in cache mode, use max 256 threads	OpenMP	Intel 18.2
GPU	K80	NVIDIA Kepler K80 GPU	CUDA	NVCC 9.2
GPU	P100	NVIDIA Pascal P100 GPU	CUDA	NVCC 9.2



(a) Time for each architecture, showing MTTKRP is the most expensive operation. (Using 100% thread capacity.)



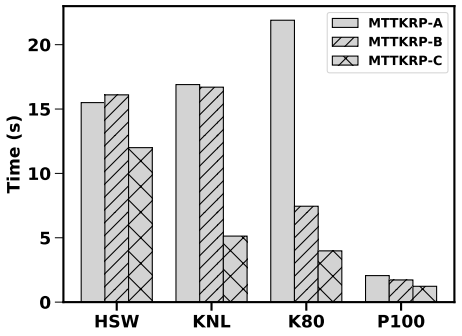
(b) For Haswell CPU and KNL platforms, showing time as the number of threads increases to the maximum available. (The number of threads used cannot be varied on GPU platforms.)

FIG. 5. *Total run-times on different architectures for ten iterations of CP-ALS using MTTKRP-B on a tensor of size* $30K \times 40K \times 50K$ *with* $10M$ *nonzeros and* $R = 128$.

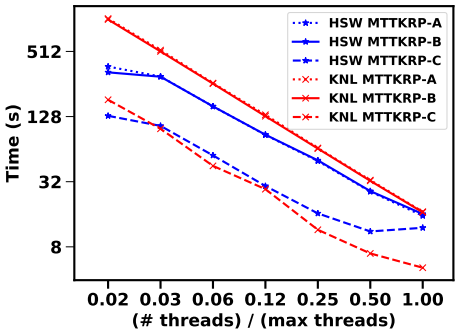
single Haswell core, we see about a 20-fold speedup on Haswell and KNL, a 38-fold speedup on the K80, and a 147-fold speedup on the P100. For the Haswell and KNL architectures, Figure 5b varies the number of threads up to the maximum number used (64 and 256, respectively), and plots the total run-time in seconds against the fraction of the total number of threads used.[7] These plots demonstrate good thread scalability on the Haswell and KNL architectures. For the GPU architectures, it is not possible to vary the number of threads.

In Figure 6, on the same data tensor, we compare MTTKRP-A (Figure 1), MTTKRP-B (Figure 3), and MTTKRP-C (Figure 4) in terms of total MTTKRP time over ten iterations of CP-ALS. We see that the use of the polymorphic arrays (MTTKRP-B) has little effect on performance for Haswell and KNL in comparison to the base MTTKRP-A, but substantially improves performance on the K80 GPU. Furthermore, we see the permutation-based approach (MTTKRP-C) is faster on all

[7] On both architectures these results were generated with `OMP_PROC_BIND=close` and `OMP_PLACES=threads` OpenMP thread-binding environment variables set.



(a) Time for each architecture, showing MTTKRP-C is faster in every case. (Using 100% thread capacity.)



(b) For Haswell CPU and KNL platforms, showing time as the number of threads increases to the maximum available. (The number of threads used cannot be varied on GPU platforms.)

FIG. 6. *Comparison of total run-times for all three MTTKRP algorithms, on different architectures, for ten iterations of CP-ALS on a tensor of size* $30K \times 40K \times 50K$ *with* $10M$ *nonzeros and* $R = 128$.

architectures, substantially so on the Haswell, KNL, and K80 architectures, but with not much difference on the P100. Clearly, atomic-write throughput is a bottleneck in MTTKRP performance, so MTTKRP-C is an improvement. We see little difference for the P100 because it has very fast double-precision atomic throughput. These timings do not include the additional setup cost associated with the permuted algorithm to compute the d permutation arrays, which is investigated later in Table 6.

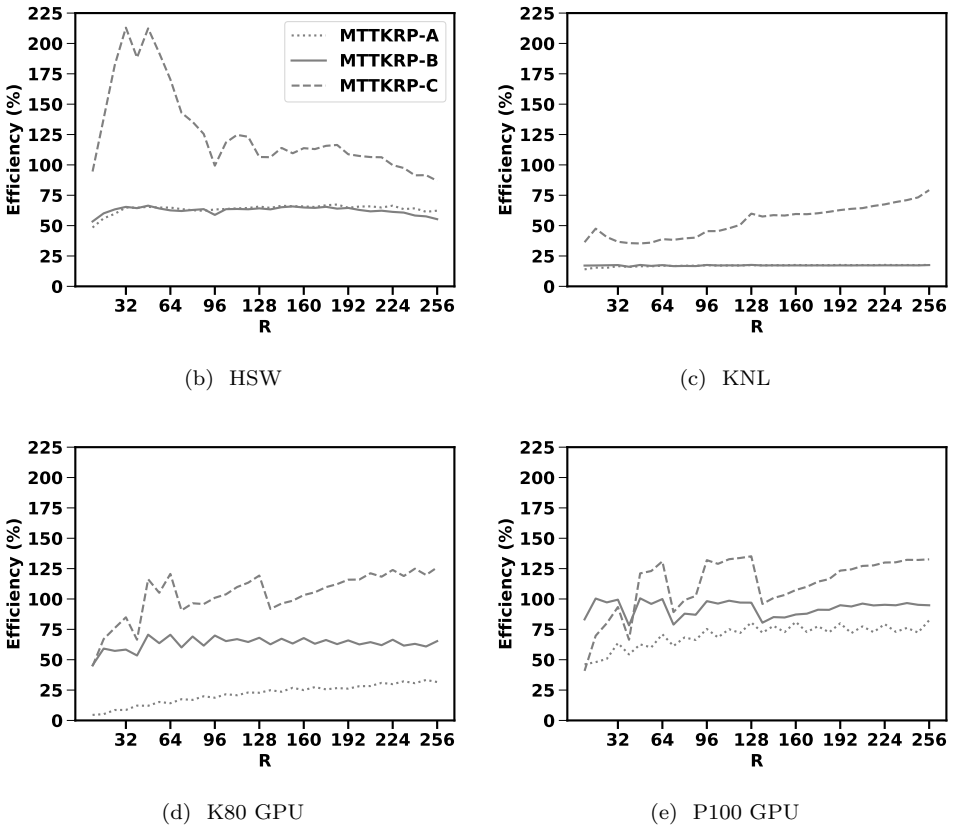
To provide quantitative evidence that the above MTTKRP algorithms are implemented efficiently, we compare their performance to a metric derived from a simplistic memory bandwidth analysis. If we assume all tensor and factor components values are read from main memory with no caching (and hence no reuse of factor matrix entires) and no thread contention for atomic-writes, then MTTKRP is a bandwidth-limited computation driven by the peak bandwidth of each architecture. To determine the peak bandwidth, we used the STREAM Triad benchmark [23] with the results shown in Figure 7a. Under these assumptions, we estimate the bandwidth for MTTKRP as

$$\frac{((dR+3)s_r + ds_o)P}{t}, \tag{7.1}$$

where s_r is the size (in bytes) of the floating-point type, s_o is the size of the ordinal type, P is the number of tensor nonzeros, and t is the total time of the MTTKRP operation. The measured bandwidth for all three versions of MTTKRP on all four architectures for $R \in [8, 256]$ with a stride of 8 is shown in Figure 7 as a percentage of the peak bandwidth in Figure 7a. For all four architectures, the bandwidth efficiency of MTTKRP-C is better since the cost of the atomic-writes is substantially reduced, making the estimate a better model of the actual MTTKRP calculation. The measured bandwidth can be greater than 100% because of cached reads of the factor matrix entries (even on the GPU architectures, some of the data reads are automatically cached in the L2 and constant caches). For KNL, the bandwidth efficiency only approaches 100% for very large R and is indicative of the general difficulty of achieving full memory bandwidth on this architecture. For the GPU architectures, we also

Architecture	Bandwidth (GB/s)
HSW	96
KNL	294
K80	177
P100	544

(a) Measured peak bandwidth for each architecture, as determined by STREAM Triad.



(b) HSW

(c) KNL

(d) K80 GPU

(e) P100 GPU

FIG. 7. *MTTKRP memory bandwidth as a percentage of peak bandwidth. The tensor is of size* $30K \times 40K \times 50K$ *with* $10M$ *nonzeros, and we vary the number of factor components* R *(x-axis). The computation used* 64*-bit floating-point and ordinal values.*

see the efficiency can be quite sensitive to R, providing the best performance when R is a multiple of 32, which is due to the methodology for computing the vector and factor matrix tile sizes in Table 4. Finally, we see the use of `TinyVec` in MTTKRP-B substantially improves performance as compared to MTTKRP-A for the GPU architectures by allowing for larger factor matrix block sizes. No improvement is seen on HSW and KNL, primarily because the original algorithm is clearly limited by atomic contention rather than memory throughput.

7.2. Real-world data and comparisons. We study the MTTKRP performance on several tensors available from the Formidable Repository of Open Sparse

Tensor and Tools (FROSTT) [28]. We selected tensors to be as large as possible, so long as they still fit within the limited memory available on the K80 GPU (12 GB). A summary of the tensors and their sizes is given in Figure 8a.

On the Haswell and KNL architectures, we compare two versions of our method (MTTKRP-B and MTTKRP-C) to the state of the art, SPLATT's CSF-based algorithms [29], and DFacTo [5]. We use two versions of SPLATT: mutexes (SPLATT-M) and tiling without mutexes (SPLATT-T), both with the default of two CSF modes. On the P100 GPU, we also compare to a highly optimized MTTKRP implementation available in ParTI! [20]. This is a COO-based MTTKRP algorithm using atomic-add to resolve race conditions in factor matrix updates. Since ParTI! implements MTTKRP only for 3- or 4-way tensors, we only include comparisons for the Uber, Enron, NELL2, and Delicious3 tensors. Furthermore, ParTI! requires hardware floating-point atomic instructions which are not available in double precision on the K80.

Figure 8b shows the results based on the total MTTKRP time for ten CP-ALS iterations with $R = 16$ factor components. The timing results shown here are aggregated over all modes of the tensor, even though significant variation in performance is often observed for different modes of a given tensor. While performance of the COO-based MTTKRP is relatively insensitive to the length of each mode (since it parallelizes over nonzeros and not factor matrix rows), it is quite sensitive to the ordering of nonzero coordinates in each mode (since this affects the amount of atomic contention). However, the permutation-based approach by design attempts to mitigate this.

If we compare just MTTKRP-B and MTTKRP-C, then MTTKRP-C (the permuted approach) is up to two orders of magnitude faster. The greater differences for real-world data tensors is due to higher atomic contention. This is most severe on the K80 architecture due to its lack of double-precision atomic instructions; conversely, it is least severe on the P100 architecture because of its support for double-precision atomic instructions. Unfortunately, the cost of storing the permutations exhausted the global memory on the K80 for the largest tensor (with 140M nonzeros).

For the tensors that ParTI! could run, we see very similar performance between ParTI! and MTTKRP-B, where MTTKRP-B is at most about 10% slower. This demonstrates the performance portability of the Kokkos implementation of MTTKRP-B, since ParTI! implements a very similar algorithm using raw CUDA and is highly tuned for the GPU.

If we compare to SPLATT on HSW, our permuted approach is faster in one case and never more than six times slower. On KNL, our approach is faster in four out of the six cases, and never worse than three times slower. Comparing to the CSR-based DFacTo, both SPLATT and our permuted approach are substantially faster. Overall, we claim that the performance of our code is comparable to that of SPLATT and ParTI! (i.e., same order of magnitude) while having the advantage of being portable. These results are summarized in Figure 8c, which displays the speedup of the best GenTen MTTKRP algorithm over the best SPLATT, DFacTo, and ParTI! algorithms.

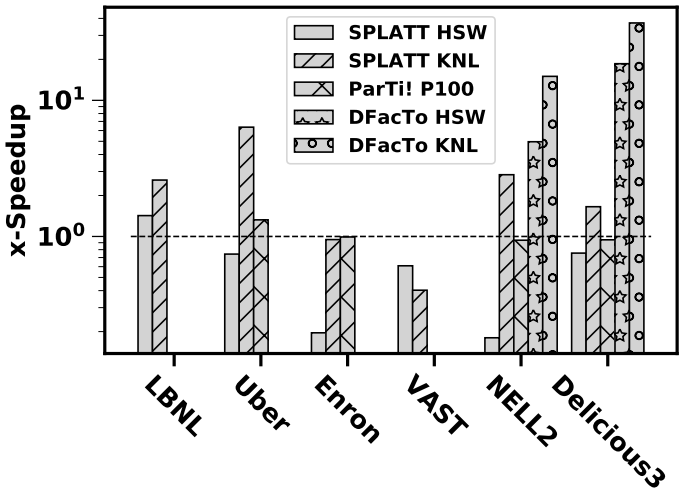
The permutation has preprocessing cost, just as SPLATT has a preprocessing cost for its CSF data structures. We show the cost of the sorting time required for the MTTKRP-C, scaled by the average CP-ALS iteration time using MTTKRP-C, for each tensor and architecture in Table 6. As described above, we use Thrust for the sorting on the GPU architectures and the OpenMP-based Intel Parallel Stable Sort on Haswell and KNL. The sorting time is less than the cost of ten CP-ALS iterations, which is relatively small and usually worthwhile given the improvement on architectures that are sensitive to atomic-writes (i.e., HSW, KNL, K80).

Name	Order	Dimensions	Nonzeros
LBNL	5	1.6K × 4.2K × 1.6K × 4.2K × 868K	1.7M
Uber	4	183 × 24 × 1.1K × 1.7K	13M
Enron	4	6.0K × 5.7K × 244K × 1.2K	54M
VAST	5	165K × 11K × 2 × 100 × 89	26M
NELL2	3	12K × 9.1K × 29K	77M
Delicious3	3	532K × 17M × 2.5M	140M

(a) Real-world tensor data sets from FROSTT.

Arch.	Method	LBNL	Uber	Enron	VAST	NELL2	Delicious3
HSW	MTTKRP-B	16.6	8.3	45.7	284.0	42.8	48.5
	MTTKRP-C	**0.4**	0.3	5.7	6.3	7.0	18.2
	SPLATT-M	0.8	**0.2**	**1.1**	26.1	**1.3**	**13.7**
	SPLATT-T	0.5	1.6	10.4	**3.8**	1.4	31.6
	DFacTo	–	–	–	–	34.6	338.3
KNL	MTTKRP-B	41.2	9.2	108.0	1910.0	82.4	77.8
	MTTKRP-C	**0.4**	**0.2**	2.8	16.1	**3.2**	**10.0**
	SPLATT-M	2.0	1.2	**2.6**	150.5	9.0	16.5
	SPLATT-T	1.0	5.1	41.3	**6.5**	15.6	97.2
	DFacTo	–	–	–	–	47.4	370.2
K80	MTTKRP-B	72.4	5.6	218.0	2010.0	55.2	**30.7**
	MTTKRP-C	**0.5**	**0.5**	**9.4**	**7.4**	**11.6**	–
P100	MTTKRP-B	0.4	0.2	2.0	4.1	1.6	2.9
	MTTKRP-C	**0.1**	**0.1**	**1.7**	**1.5**	1.8	7.0
	ParTI!	–	0.3	1.9	–	**1.5**	**2.8**

(b) Total MTTKRP time (in seconds) for ten iterations with R=16 for each data tensor, architecture, and method. Due to the increased storage requirements of the permuted approach, global memory was exhausted for MTTKRP-B for the Delicious3 tensor on the K80 architecture. DFacTo implements MTTKRP only for 3-way tensors, and ParTI! implements MTTKRP only for 3- or 4-way tensors.



(c) Speedup of the best GenTen MTTKRP method over the best SPLATT, DFacTo, and ParTI! methods.

FIG. 8. *Results on real-world tensor data sets.*

TABLE 6
Sorting cost for the permutation-based MTTKRP approach scaled by the average (permuted) CP-ALS iteration time with R=16.

Architecture	LBNL	Uber	Enron	VAST	NELL2	Delicious3
HSW	0.6	4.3	9.2	5.3	6.2	2.8
KNL	1.0	5.5	8.6	1.3	7.6	4.6
K80	1.1	3.0	3.3	3.3	2.6	–
P100	0.5	1.3	4.2	3.6	4.6	2.2

8. Conclusions. In this paper we describe a portable and performant implementation of MTTKRP for sparse tensors on emerging computer architectures, including multicore CPUs, manycore Intel Xeon Phis, and NVIDIA GPUs. For a sparse tensor stored in coordinate format, we showed how to arrange the loops to achieve fine-grained parallelism, in Figure 1. The portable implementation is primarily facilitated by the Kokkos library, which provides data structures and abstractions that enable performance on multiple architectures. One of the complications of our implementation is that each thread requires its own temporary storage. Such an allocation is limited in Kokkos, so we introduced `TinyVec`, an extension of the Kokkos framework for polymorphic data arrays that stores the temporary data in registers and whose length is parameterized by an architecture-dependent compile-time constant. The resulting algorithm is shown in Figure 3. To avoid atomic operations, we do some preprocessing so that we can loop through the coordinates of each mode in increasing order through the use of *permutation arrays*. This doubles the storage for the indices, but the increase in performance for the version shown in Figure 4 can be considerable. Our implementation of CP-ALS using our improvements to MTTKRP is available in the open-source software package GenTen.

We studied the performance of GenTen on a variety of contemporary architectures and demonstrated that GenTen's MTTKRP is efficient on all of these architectures by comparing to the expected computational bandwidth. We also compared the performance of GenTen's MTTKRP to SPLATT and DFacTo on CPU and KNL platforms and ParTI! on the P100 GPU using several realistic data tensors from FROSTT, demonstrating comparable or better performance.

Future work with GenTen will involve the incorporation of distributed memory parallelism to enable analysis of larger tensors, as well as generalization of the algorithms to be applicable to more general categories of data tensors (such as count and binary data) using the generalized CP method introduced in [12]. We will also investigate approaches for performing the necessary tensor sorting operations required by the permutation-based MTTKRP algorithm while the tensor is being read from disk to eliminate this extra cost (by, e.g., having one thread read the data from disk and have one or more threads sort it). Following some of the ideas in SPLATT [30], performance on KNL architectures can be improved by placing the factor matrices in HBM memory (and putting the KNL in so-called flat mode) and most other data in main memory. (This optimization does not appear to be present in the public version of SPLATT used in our results.) Results in [30] suggest this could improve performance by as much as a factor of two. We are also exploring other approaches to addressing atomic contention within MTTKRP, such as the use of thread-private copies of the resulting factor matrix that are later reduced across threads (similar to the approach presented in [30]). Preliminary work in this direction suggests it can be beneficial to the coordinate-based MTTKRP algorithm, as long as the number

of rows of the factor matrix or the number of threads is not too large. Ultimately, we are likely to converge to a hybrid approach that combines atomic and reduction approaches.

Acknowledgments. We thank the anonymous referees for critical feedback on this work, resulting in substantial improvements in the presentation. We also thank Dr. Jiajia Li for the use of her ParTI! code in the numerical results.

REFERENCES

[1] B. W. Bader and T. G. Kolda, *Algorithm* 862: *MATLAB tensor classes for fast algorithm prototyping*, ACM Trans. Math. Software, 32 (2006), pp. 635–653, https://doi.org/10.1145/1186785.1186794.

[2] B. W. Bader and T. G. Kolda, *Efficient MATLAB computations with sparse and factored tensors*, SIAM J. Sci. Comput., 30 (2007), pp. 205–231, https://doi.org/10.1137/060676489.

[3] N. Bell and M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, ACM, New York, 2009, 18, https://doi.org/10.1145/1654059.1654078.

[4] J. D. Carroll and J. J. Chang, *Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition*, Psychometrika, 35 (1970), pp. 283–319, https://doi.org/10.1007/BF02310791.

[5] J. H. Choi and S. Vishwanathan, *DFacTo: Distributed factorization of tensors*, in Advances in Neural Information Processing Systems (NIPS 2014), 2014, pp. 1296–1304, http://papers.nips.cc/paper/5395-dfacto-distributed-factorization-of-tensors.

[6] N. Cohen, O. Sharir, and A. Shashua, *On the expressive power of deep learning: A tensor analysis*, in 29th Annual Conference on Learning Theory, Vol. 49, Proceedings of Machine Learning Research, 2016, pp. 698–728, http://proceedings.mlr.press/v49/cohen16.html.

[7] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, *Manycore performance-portability: Kokkos multidimensional array library*, Sci. Program., 20 (2012), pp. 89–114, https://doi.org/10.3233/SPR-2012-0343.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland, *Kokkos: Enabling manycore performance portability through polymorphic memory access patterns*, J. Parallel Distrib. Comput., 74 (2014), pp. 3202–3216, https://doi.org/10.1016/j.jpdc.2014.07.003.

[9] H. Fanaee-T and J. Gama, *Tensor-based anomaly detection: An interdisciplinary survey*, Knowledge-Based Systems, 98 (2016), pp. 130–147, https://doi.org/10.1016/j.knosys.2016.01.027.

[10] E. Gujral, R. Pasricha, and E. E. Papalexakis, *SamBaTen: Sampling-based Batch Incremental Tensor Decomposition*, preprint, https://arxiv.org/abs/1709.00668v1, 2017.

[11] R. A. Harshman, *Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multimodal factor analysis*, UCLA Working Papers in Phonetics, 16 (1970), pp. 1–84; available at http://www.psychology.uwo.ca/faculty/harshman/wpppfac0.pdf.

[12] D. Hong, T. G. Kolda, and J. A. Duersch, *Generalized Canonical Polyadic Tensor Decomposition*, preprint, https://arxiv.org/abs/1808.07452, 2018.

[13] M. Janzamin, H. Sedghi, and A. Anandkumar, *Beating the Perils of Non-convexity: Guaranteed Training of Neural Networks using Tensor Methods*, preprint, https://arxiv.org/abs/1506.08473v3, 2016.

[14] P. Karpiński and J. McDonald, *A high-performance portable abstract interface for explicit SIMD vectorization*, in Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'17, ACM, New York, 2017, pp. 21–28, https://doi.org/10.1145/3026937.3026939.

[15] O. Kaya and B. Uçar, *Parallel CANDECOMP/PARAFAC decomposition of sparse tensors using dimension trees*, SIAM J. Sci. Comput., 40 (2018), pp. C99–C130, https://doi.org/10.1137/16M1102744.

[16] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, *Designing vector-friendly compact BLAS and LAPACK kernels*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, 2017, 55, https://doi.org/10.1145/3126908.3126941.

[17] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500, https://doi.org/10.1137/07070111X.

[18] M. Kretz and V. Lindenstruth, *Vc: A C++ library for explicit vectorization*, Software: Practice and Experience, 42 (2012), pp. 1409–1430, https://doi.org/10.1002/spe.1149.

[19] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, *Model-driven sparse CP decomposition for higher-order tensors*, in Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 1048–1057, https://doi.org/10.1109/IPDPS.2017.80.

[20] J. Li, Y. Ma, and R. Vuduc, *ParTI!: A Parallel Tensor Infrastructure for Multicore CPUs and GPUs*, Oct. 2018, https://github.com/hpcgarage/ParTI.

[21] J. Li, J. Sun, and R. Vuduc, *HiCOO: Hierarchical Storage of Sparse Tensors*, in Proceedings of the ACM/IEEE International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC18), IEEE Press, Piscataway, NJ, 2018, 19.

[22] B. Liu, C. Wen, A. D. Sarwate, and M. Mehri Dehnavi, *A Unified Optimization Approach for Sparse Tensor Operations on GPUs*, preprint, https://arxiv.org/abs/1705.09905, 2017.

[23] J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 12 (1995), pp. 19–25.

[24] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, *Tensorizing Neural Networks*, preprint, http://arxiv.org/abs/1509.06569, 2015.

[25] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 2013, https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[26] E. Phipps, M. D'Elia, H. C. Edwards, M. Hoemmen, J. Hu, and S. Rajamanickam, *Embedded ensemble propagation for improving performance, portability, and scalability of uncertainty quantification on emerging computational architectures*, SIAM J. Sci. Comput., 39 (2017), pp. C162–C193, https://doi.org/10.1137/15M1044679.

[27] T. B. Rolinger, T. A. Simon, and C. D. Krieger, *Performance challenges for heterogeneous distributed tensor decompositions*, in IEEE High Performance Extreme Computing Conference HPEC 2017, IEEE, 2017.

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

[29] 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, IA3 '15, ACM, New York, 2015, 5, https://doi.org/10.1145/2833179.2833183.

[30] S. Smith, J. Park, and G. Karypis, *Sparse tensor factorization on many-core processors with high-bandwidth memory*, in Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 1058–1067, https://doi.org/10.1109/IPDPS.2017.84.

[31] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, *SPLATT: Efficient and parallel sparse tensor-matrix multiplication*, in IPDPS 2015: IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 61–70, https://doi.org/10.1109/ipdps.2015.27.

[32] E. Solomonik, D. Matthews, J. R. Hammond, J. F. Stanton, and J. Demmel, *A massively parallel tensor contraction framework for coupled-cluster computations*, J. Parallel Distrib. Comput., 74 (2014), pp. 3176–3190, https://doi.org/10.1016/j.jpdc.2014.06.002.

[33] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, Reading, MA, 2002.

[34] H. Wang, P. Wu, I. G. Tanase, M. J. Serrano, and J. E. Moreira, *Simple, portable and fast SIMD intrinsic programming: Generic simd library*, in Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP '14, ACM, New York, 2014, pp. 9–16, https://doi.org/10.1145/2568058.2568059.

[35] 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, KDD '15, ACM, New York, 2015, pp. 1265–1274, https://doi.org/10.1145/2783258.2783395.