SOFTWARE FOR SPARSE TENSOR DECOMPOSITION ON EMERGING COMPUTING ARCHITECTURES∗
ERIC PHIPPS† AND TAMARA G. KOLDA‡

Abstract. In this paper, we describe a new software package for sparse tensor decompositions that targets emerging parallel computing architectures. We are focused on the canonical polyadic tensor decomposition of sparse tensors, whose key kernel is the matricized tensor times Khatri-Rao product (MTTKRP). Our goal is to develop software that is portable to a variety of multicore and manycore computing architectures, such as multicore CPUs, the Intel Xeon Phi, and NVIDIA GPUs. To do this, we use the Kokkos framework, which is designed to enable high performance across multiple architectures with a single code even though these architectures handle fine-grained parallelism in varying ways. Not only are the specifics of the implementation interesting to persons tuning tensor computations, but also potentially to other high-performance software developers looking to understand ways to make their codes portable across architectures without supporting multiple code bases. Kokkos is an evolving but extensible system, and we explain one problem we had in achieving performance and how we surmounted it by encapsulating fine-grained parallelism through the use of a special compile-time polymorphic array type. We also introduce a new sparse tensor format based on a variation of the traditional coordinate storage format. This format reorders the traversal of tensor nonzeros in the MTTKRP to reduce atomic-write contention; this new format leads to significantly improved performance on all architectures with minimal increase in memory footprint. Performance of our software is measured on several computing platforms, including the Xeon Phi and NVIDIA GPUs. We show that we are competitive with state-of-the-art approaches available in the literature while having the advantage of being able to run on a wider of variety of architectures with a single code.

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

1. Introduction. Tensors, or multidimensional arrays, are a powerful means of representing relationships in multiway data; see the survey paper [13]. We focus on the canonical polyadic (CP), also known as CANDECOMP/PARAFAC, decomposition [11, 4] of sparse tensors. The CP decomposition is a low-rank decomposition and represents approximates a given tensor by a sum of rank-one tensors. CP decompositions have numerous applications in data science, including analysis of online social networks [10], anomaly detection [9], compression of neural nets [12, 20, 6], health data analytics [28], among others.

In this work, we consider efficient computation of the CP decomposition of sparse tensors on emerging architectures. Our focus is on the alternating least squares (CP-ALS) method for sparse tensors, and the main computational kernel of interest is the so-called matricized tensor times Khatri-Rao product (MTTKRP). We describe the mathematical background on tensors and MTTKRP in section 2.

Our goal is to develop software that is portable to a variety of multicore and manycore computing architectures, such as multicore CPUs, the Intel Xeon Phi (manycore), and NVIDIA GPUs. Our library leverages the package Kokkos [7, 8] to allow a single implementation to be portable to a diversity of shared memory parallel programming

∗This work was supported by the Laboratory Directed Research and Development 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. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.
†Sandia National Laboratories, Albuquerque, NM (etphipp@sandia.gov).
‡Sandia National Laboratories, Livermore, CA (tgkolda@sandia.gov).
models (such as OpenMP, pThreads, and CUDA) and hardware. Developing a single parallel implementation that is portable to and performant on a diversity of architectures is complicated by the manner in which architectures express fine-grained parallelism. 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, reviewed in section 3, provides abstractions that unify these different approaches, making a single software implementation feasible.

In this work, we present the first portable and performant Kokkos-based software implementations of the alternating least squares (CP-ALS) method for sparse tensors, which is available in the open source GenTen library. 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. In section 4, we describe how to use the Kokkos abstractions within the context of the key MTTKRP kernel of CP-ALS.

Kokkos is an evolving but extensible system, and we explain one problem in achieving performance and how we surmount it using a new Kokkos-like abstraction based on compile-time polymorphic arrays in section 5. We contrast this approach to the Kokkos scratch pad arrays. The result is that our method leads to higher performance and, we contend, is simpler to use than what was already available in Kokkos.

We also develop a new data structure for storing sparse tensors that is amenable to high performance, described in section 6. To avoid race conditions in thread-parallel writes, we rely on atomic write instructions provided through Kokkos. These atomic instructions substantially reduce performance for poorly-structured tensors, so a modification of the coordinate-based tensor format is introduced that reduces write contention in MTTKRP.

Performance results on multicore CPUs, NVIDIA GPUs, and the Knights Landing (KNL) version of the Intel Xeon Phi are provided in section 7, along with performance comparisons with a state-of-the-art open-source code called SPLATT [25]. These results show that GenTen achieves state-of-the-art performance on a variety of platforms with a single codebase. We discuss conclusions and future work in section 8.

1.1. Related work. SPLATT [25] 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 increases memory costs. More recently, SPLATT incorporated a compressed sparse fiber (CSF) approach [23] that uses a tree-based representation and avoids duplication of the tensor for each mode. The resulting 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 [24], where several variants of the MTTKRP

---

1https://gitlab.com/tensors/genten
algorithm were considered.

Building on the CSF data structure, Li et al. proposed an adaptive tensor memo-
ization 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 [14] due to storing semi-sparse intermi-
nate 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 pro-
posed but only studied within the context of thread parallelism on multicore CPU
architectures.

DFacTo [5] stores a sparse tensor as a set of sparse matrices and can rely on op-
timized implementations of sparse matrix-vector products to implement MTTKRP.
Similar to the earlier SPLATT implementation, a different copy of the tensor is re-
quired for each mode. DFacTo only considers distributed memory parallelism of the
MTTKRP for third-order tensors.

The Cyclops Tensor Framework [26] provides a general framework for implement-
ing 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. The library is geared towards distributed memory parallelism.

More recently, Liu et al. proposed an extension of coordinate-based tensor storage
format called F-COO (flagged-coordinate) [18] which is extensible to many types of
tensor operations, and demonstrated CP decomposition results on GPUs.

Finally, Li et al. proposed a variant of the COO format called HiCOO [17] that
decomposes a sparse tensor into sparse blocks, reducing the memory required to store
tensor nonzeros (and hence memory bandwidth to read them). A two-level thread par-
allelization scheme for MTTKRP leveraging the block structure is also investigated,
demonstrating speedup over both the COO and CSF formats.

1.2. Contributions. This work provides three primary contributions. First and
foremost, this work describes how to achieve performance portability for the MT-
TKRP kernel by using Kokkos to exploit fine-grained parallelism. In the coarse of
this adaptation, we introduce a novel approach for obtaining portable performance in
the MTTKRP calculation through the use of thread-local, compile-time polymorphic
arrays. Second, we present a new sparse tensor storage format that dramatically re-
duces computational cost of the MTTKRP algorithm on a variety of contemporary
architectures and is shown to provide comparable or better performance than other
state-of-the-art approaches. Third, our publicly available GenTen software library
represents the first-ever implementation of all critical CP-ALS kernels in a portable
manner, yet still achieves high performance with the same algorithmic implementation
on CPUs, Intel Xeon Phi, and NVIDIA GPUs.

2. CP tensor decomposition. Except for specific relevant concepts discussed
in detail later in this section, we assume a basic familiarity with tensors and refer the
reader to [13] for further details. Let $X \in \mathbb{R}^{N_1 \times \cdots \times N_d}$ be a given $d$-way tensor. We
refer to each way or dimension as a mode. For a given $R$, the goal is to find a low-rank
model tensor $M$ that is a good approximation to $X$, i.e.,

\[
\min_M \|X - M\| \quad \text{s.t.} \quad M = \sum_{j=1}^{R} \lambda_j a_j^{(1)} \circ a_j^{(2)} \circ \cdots \circ a_j^{(d)},
\]
where $\lambda_j$ is a scalar weight, $\mathbf{a}_j^{(n)}$ is a column vector of size $N_n$ that is assumed to be normalized to length one in some norm, and $\circ$ represents the tensor outer product. For notation convenience, we assemble all the column vectors in mode $n$ into a factor matrix of size $N_n \times R$:

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

Hence, the goal is to find the weight vector $\mathbf{\lambda} = [\lambda_1 \cdots \lambda_R]^\top$ and the $d$ factor matrices $\{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(d)}\}$ that define the low-rank model tensor $\mathbf{M}$.

In this work, we compute the CP decomposition using the alternating least squares (CP-ALS) method [11, 4]. Details are omitted here but can be found in, e.g., the survey by Kolda and Bader [13]. Our main interest is in the matricized tensor times Khatri-Rao product (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 $i$th nonzero and its subscripts as $x_i$ and $(\ell_{i1}, \ell_{i2}, \ldots, \ell_{id})$. If there are $P$ nonzeros, then we store $\mathbf{X}$ with a $P$-vector of real values and a $P \times d$ vector of coordinates.

For a sparse tensor, the mode-$n$ MTTKRP computes a matrix $\mathbf{V}$ of size $N_n \times R$ that is defined elementwise as

$$
v(k,j) = \lambda_j \sum_{i=1}^P \mathcal{X}_{x_i} \prod_{m=1}^d a^{(m)}(\ell_{im},j) \quad \text{for} \quad k = 1, \ldots, N_n \text{ and } j = 1, \ldots, R.
$$

We assume $d$ is small, ranging from 3 to 5 in the examples we show. In order of magnitude, $P$ ranges from $10^6$–$10^8$, $N_n$ 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. A simple implementation is shown in Figure 1.

3. Kokkos for manycore architectures. Kokkos [7, 8] is 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 such as multicore CPUs, the Intel Xeon Phi, and NVIDIA GPUs. Kokkos defines several abstractions that are used in the implementation of parallel algorithms:

- **Parallel Pattern:** Computational execution pattern, such as `parallel_for`, `parallel_reduce`, or `parallel_scan`.
- **Execution Policy:** How to execute, including choices such as dynamic scheduling or teams of threads working in conjunction over an iteration space.
- **Execution Space:** Where a computation executes, e.g., on a CPU or a GPU. For a CPU, which NUMA regions and/or which cores will be used.
- **Memory Space:** Where data resides, such as host memory, GPU memory, high-bandwidth memory, and so on.
- **Memory Layout:** Data arrangement in memory, such as row major, column major, or tiled for a two-dimensional array of data.

By tightly coordinating the mapping of these abstractions to the computer architecture capabilities, high performance and thread scalability can be achieved through a single code body for each distinct parallel algorithm.

The fundamental data structure within Kokkos is the `View`, which stores a multidimensional array of data. The template parameters specify the type of the data.
void mttkrp(const Sptensor & X, const Ktensor & M,
             const unsigned n, FacMatrix & V) {
    const size_t nnz = X.nnz(); // number of nonzeros in X
    const unsigned d = u.ndims(); // number of modes in X
    const unsigned R = u.ncomponents(); // number of component in M

    for (size_t i = 0; i < nnz; ++i) { // iterate over nonzeros
        const size_t k = X.subscript(i,n); // mode-n subscript of i-th nonzero
        const double x_val = X.value(i); // i-th nonzero
        for (unsigned j = 0; j < R; ++j) { // iterate over factor matrix columns
            double tmp = x_val * M.weights(j); // M.weight(j) returns j-th weight
            for (unsigned m = 0; m < d; ++m) { // iterate over factor matrices
                if (m != n) { // skip n-th factor matrix
                    tmp *= M[m].entry(X.subscript(i,m),j); // M[m] returns m-th factor matrix
                }
            }
            V.entry(k,j) += tmp; // accumulate contribution to row k of result
        }
    }
}

Fig. 1: Simple serial implementation of MTTKRP for a sparse tensor X times the
model M along mode n, with the result stored in the matrix V.

(e.g., float or double), its number of dimensions, the space, and the layout. The
space may be a memory space, an execution space (in which case data is allocated
in the default memory space for that execution space), or a memory-execution space
pair. This allows GenTen data structures to be allocated in different memory spaces
(e.g., CPU or GPU memory), and functions are provided to transfer these data struc-
tures between memory spaces. The layout is the only optional parameter and is
chosen automatically if unspecified. For a given parallel algorithm, the user selects
an appropriate parallel pattern, e.g., parallel for for a matrix-vector product or
parallel reduce for an inner product. The user provides the parallel pattern with a
“policy” that specifies the iteration space (among other details) and either a functor
or lambda-expression that is invoked on each element of the iteration space. Recall
that a functor is a C++ class that overloads operator(), and a lambda expression
is an anonymous functor created by the C++ compiler using special syntax. Kokkos
provides the boilerplate code for a functor via KOKKOS_LAMBDA, a preprocessor macro
that expands to a capture-by-value lambda ([=]) with additional attributes to allow
the lambda to execute on a GPU. In both cases, the functor/lambda just serves as
the mechanism for encapsulating the code to be executed in parallel.

To achieve portability, the code body must be agnostic to the execution and
memory spaces, allowing the same code body to be compiled for multiple architec-
tures. For high performance, the execution policy and memory layout must be tightly
integrated. In many cases, the user can rely on Kokkos to make intelligent choices au-
tomatically, but the user also has the option to specify policies for parallelization (e.g.,
grouping the threads) or specify appropriate data layouts. For instance, a user can
explicitly specify LayoutRight to specify that a matrix be stored row-wise, which we
do for the CP factor matrices. Most modern architectures allow grouping of threads
in a hierarchical fashion to take advantage of shared hardware resources. On CPUs
and the Intel Xeon Phi, each physical core has two or more hyperthreads that share L1 cache. Each hyperthread can execute vector instructions, i.e., Single Instruction Multiple Data (SIMD). Different groups of cores share higher-level caches and corresponding different costs associated with accessing different regions of main memory, i.e., comprising different NUMA regions. For a schematic diagram of a typical CPU architecture, see Figure 2a (an Intel Xeon Phi is conceptually similar).

GPU architectures consist of many streaming multiprocessors (SMs), each comprising a large number of scalar cores. The NVIDIA CUDA programming model organizes a large number of logical threads into a grid of thread blocks, where each thread block consists of a collection of threads. Each thread within a thread block executes on a separate scalar core. Therefore, while threads within a thread block are individually programmed, they are organized into groups called warps that execute the same instruction at every clock cycle, i.e., Single Instruction Multiple Thread (SIMT). Inactive threads within a warp, due to branching for example, are masked off. Some models of GPUs have a small L1 cache shared by all threads within a thread block, and most have a small, fast scratchpad memory called shared memory that is addressable inside a parallel kernel and is used to share or communicate data between threads within a thread block. All global memory accesses go through an L2 cache that is shared across the whole device. Other caches such as texture cache or constant cache may be used by the compiler automatically. Threads within a thread block are synchronized, whereas thread blocks are intended to execute independently. See Figure 2b for a schematic diagram of a GPU architecture.

While the programming models for these architectures are quite different, it is possible to map the hardware capabilities between the two. The threads within a GPU warp conceptually are similar to the vector lanes in a CPU architecture, with the distinction that vector lanes are not individually programmable. Similarly, the collection of warps in a thread block can be viewed as analogs to the hyperthreads on one or more cores. Using this mapping, Kokkos provides execution policies to exploit hierarchical parallelism in a portable manner. Kokkos uses terminology mirroring the nested parallelism concepts introduced in the OpenMP 4.0 specification [21] 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. Using the above mapping, a team within the league corresponds to a collection of one or more hyperthreads on a CPU architecture or a thread block on a GPU architecture, while vector parallelism corresponds to CPU vector instructions or threads within a GPU warp.

To use the team-based execution policies, the required league, team, and vector sizes are supplied by the user via Kokkos::TeamPolicy. The league size defines the iteration space that the parallel pattern iterates over. The parallel pattern functions (e.g., parallel_for) are invoked within the code body with special execution policies to signal vector parallelism. In most cases, nested calls to the parallel pattern functions use lambda-expressions to implement vector parallelism, taking an index over some range as an argument. On a CPU/Phi architecture, these patterns are implemented as simple loops that call the lambda-expression for each loop index. These loops are intended to be auto-vectorized by the compiler, and Kokkos incorporates pragmas to aid this process. Conversely, on the GPU architecture each thread within a warp executes the supplied lambda one or more times computing its local vector index from its CUDA thread index.

The best team and vector sizes are architecture and algorithm dependent. On CPU/Phi architectures, the vector size is not used by Kokkos and can be set to one, and the team size is constrained by the topology of the CPU/Phi architecture. On
4. MTTKRP using Kokkos. In this section we describe the implementation of the MTTKRP kernel using Kokkos, including the key data structures for storing a sparse tensor and its decomposition. All GenTen data structures use Kokkos View multidimensional array classes for storing data as described in section 3. They are template classes with a single template parameter Space, which determines where data is allocated and the CP-ALS kernels are executed.

4.1. Sparse tensor format. A variety of sparse tensor formats have been proposed in the literature [2, 25, 5, 18, 17]. The simplest is the coordinate format where the tensor $X$ is stored as list nonzero indices and values [2], and this is what we use
**Fig. 3:** Classes for storing sparse tensor, factor matrix, and Kruskal tensors. Kokkos commands are highlighted in red. For brevity, only class members needed for the MTTKRP kernel are shown. All classes are contained within the Genten namespace and have a single template parameter, Space, indicating the execution space in which kernels are executed and/or memory space where data is stored. KOKKOS_INLINE_FUNCTION is a Kokkos-supplied macro that adds the necessary attributes for the functions to be callable on both the host and device for GPU-enabled builds. The FacMatrix uses an explicit Kokkos::LayoutRight layout.

Here. As we can see in the Sptensor class in Figure 3, we store the indices as a two-dimensional array of ordinals and the nonzero values as a one-dimensional array of floating-point data. The total size of a tensor $X$ in memory is then $(d_{so} + d_{sf}) \cdot \text{nnz}(X)$ bytes where $s_{o}$ and $s_{f}$ are the sizes in bytes of the ordinal and floating-point types, respectively, and \text{nnz}(X) is the number of nonzeros in $X$.

**4.2. K-tensor format.** Following [2], we refer to the CP decomposition given by (1) as a Kruskal tensor, or K-tensor for short. It consists of a one-dimensional array of length $R$ of floating-point values to store the weights $\lambda$ and a length-$d$ array of factor matrices. Each factor matrix $A^{(n)}$ consists of a two-dimensional array of size $N_{n} \times R$ of floating-point values, stored using the Kokkos::LayoutRight (i.e., rowwise) memory layout to support the row-based MTTKRP algorithm described below. The factor matrix and K-tensor in GenTen are encapsulated in C++ classes.

\[ \text{nnz}(X) = \text{nnz}(\lambda) + \sum_{n=1}^{d} (N_{n} - 1) \cdot \text{nnz}(A^{(n)}) \]
template <typename Space>
void mttkrp(const Sptensor<Space>& X, const Ktensor<Space>& M,
            const unsigned n, const FacMatrix<Space>& V) {
    const size_t nnz = X.nnz();
    const unsigned d = M.ndims();
    const unsigned R = M.ncomponents();

    // iterate over nonzeros
    Kokkos::parallel_for(Kokkos::RangePolicy<Space>(0, nnz), // iteration range
                          KOKKOS_LAMBDA(const size_t i) { // begin lambda
        const size_t k = X.subscript(i,n);
        const double x_val = X.value(i);
        for (unsigned j =0; j<R; ++j) {
            double tmp = x_val * M.weights(j);
            for (unsigned m =0; m<d; ++m) {
                if (m != n)
                    tmp *= M[m].entry(X.subscript(i,m),j);
            }
            // accumulate contribution to row k of result
            Kokkos::atomic_add(&V.entry(k,j), tmp[j]);
        }
    }); // end lambda
}

Fig. 4: Naive conversion of the serial MTTKRP algorithm to Kokkos by parallelizing over tensor nonzeros.

called FacMatrix and Ktensor, summarized in Figure 3.

4.3. Achieving performance for MTTKRP. Developing a single, performant implementation of MTTKRP that is portable to diverse architectures is challenging due to the different mechanisms that these architectures use for expressing fine-grained parallelism. Consider the simple serial MTTKRP implementation shown in Figure 1. A naive parallel implementation using Kokkos is shown in Figure 4, where the for-loop is replaced by the Kokkos::parallel_for pattern. Here Kokkos::RangePolicy<Space>(0,nnz) indicates the parallel-for iterates over the range \{0,1,\ldots,nnz\}, executing the code body contained within the lambda expression marked by Kokkos::LAMBDa. Since threads are processing many nonzeros concurrently, multiple threads may write to the same row of V simultaneously, so these writes must be serialized. Here we do this using the atomic instruction Kokkos::atomic_add().\(^3\)

This implementation is portable and can run on any architecture that Kokkos supports. However it has several issues that limit performance. First, it does not take advantage of reuse of the tensor indices across the \(R\) columns of the factor matrices. More importantly, it does not take advantage of additional parallelism in the loop over factor matrix columns and the potential for packed/coalesced accesses of the factor matrix entries. On caching architectures such as CPUs and the Intel Xeon Phi, packed accesses refers to a given thread accessing consecutive memory locations sequentially.

\(^3\)The availability of those instructions depends upon both the hardware and the floating-point data type. For example, NVIDIA K80 GPUs have hardware atomic-add instructions for single-precision data but not double-precision. If hardware instructions are not available, Kokkos uses a general implementation through atomic compare-and-swap (CAS), which can be dramatically slower if there is high contention among threads.
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. In both cases, packed/coalesced accesses are usually required to achieve full utilization of the available memory bandwidth, and this is critical for achieving performance with MTTKRP since the performance is limited by the bandwidth of reading the factor matrices from memory.

These shortcomings are addressed by moving the factor matrix column loop to the lowest loop. This requires creating a temporary array of length $R$ to store the accumulation of the factor matrix products across tensor dimensions. Since dynamic memory allocation within a thread-parallel kernel is usually prohibitively expensive (since these allocations are typically serialized by the operating system), one approach to creating this buffer is to use the scratch pad facilities provided by Kokkos. This requires informing Kokkos of how much memory is needed per team, and then obtaining a per-team handle to this memory within the kernel. Behind the scenes, Kokkos allocates the memory needed for all teams within the scratch pad prior to kernel launch, so no dynamic memory is allocated within the kernel. On a GPU, this scratch pad memory will be allocated in the shared memory space for fast access. Parallelism across factor matrix columns is introduced through the Kokkos team-based execution policy and nested parallel-for patterns. These modifications are presented in Figure 5. Here we create a `Kokkos::TeamPolicy` with the league size given by the number of tensor nonzeros, the team size equal to one, and the vector-size equal to 16. On a CPU or Xeon Phi architecture, this is effectively the same as the range policy showed previously; but on a GPU, each team will consist of 16 CUDA threads for use in vector/warp parallelism. The lambda expression is invoked using the `TeamPolicy::member_type` team handle, which is a simple structure providing the league and team index, as well as the league and team sizes. Since the league size is the same as the number of tensor nonzeros, the league index is just the tensor nonzero index that team will process. A temporary buffer of length $R$ is created for each team, allocated in the scratch memory space, and wrapped by a `Kokkos::View` using the pointer returned by `team.team_scratch()`. The parallel body includes nested parallel-for patterns, iterating over the range $\{0, 1, \ldots, R-1\}$ for each team. On a CPU or Xeon Phi, these parallel-for loops are implemented with simple for-loops (with additional pragmas to encourage vectorization); but on a GPU, each CUDA thread will iterate over the range with a stride of `VectorSize`, allowing `VectorSize` columns to be processed in parallel for each iteration. Moreover, by using `Kokkos::LayoutRight` for the memory layout of the factor matrices (see Figure 3), the reads of the factor matrices and writes to $V$ are fully packed/coalesced.

The implementation in Figure 5 is likely to perform much better than the naive implementation in Figure 4; however, it still has issues that reduce performance. Kokkos computes the CUDA thread block size as the product of the team and vector sizes, which in this case is just 16. Since there is a limit to the number of thread blocks that can be simultaneously active, typically the block size should be in the range 128–256 to have as many threads active as possible (called occupancy). This can be fixed in a straightforward fashion by increasing the team size (to, say, 8–16) on the GPU, and modifying the kernel so that each team processes multiple tensor nonzeros. More importantly, the amount of shared memory available on GPUs is fairly small (typically 48-96 KB per SM, depending on the GPU), and this memory must be shared by all active thread blocks executing on each SM. Thus as $R$ increases, the observed performance will decline due to greater shared memory use by each team, resulting in fewer teams that can be simultaneously active on each SM. This can be
template <typename Space>
void mttkrp(const Sptensor<Space>& X, const Ktensor<Space>& M,
const unsigned n, const FacMatrix<Space>& V) {

    const size_t nnz = X.nnz();
    const unsigned d = M.ndims();
    const unsigned R = M.ncomponents();

    // define space policy that specifies team parallelism
    typedef Kokkos::TeamPolicy<Space> Policy;
    const unsigned VectorSize = 16;
    const unsigned TeamSize = 1;
    Policy policy(nnz, TeamSize, VectorSize);

    // typedef to define scratch page space for each thread
    typedef Kokkos::View<double*, typename Space::scratch_memory_space,
        Kokkos::MemoryUnmanaged> ScratchSpace;
    const size_t bytes = ScratchSpace::shmem_size(R);

    // loop over nonzeros
    Kokkos::parallel_for(policy.set_scratch_size(0, Kokkos::PerTeam(bytes)),
        KOKKOS_LAMBDA(typename Policy::member_type team) {
            const size_t i = team.league_rank();
            const size_t k = X.subscript(i,n);
            const double x_val = X.value(i);
            ScratchSpace tmp(team.team_scratch(0), R); // scratch pad

            // loop over matrix columns
            Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                [&] (const unsigned j) { // nested lambda
                    tmp[j] = x_val * M.weights(j);
                });

            // loop over factor matrices
            for (unsigned m =0; m<d; ++m) {
                if (m != n) {
                    const double *row = &M[m].entry(X.subscript(i,m),0);

                    // loop over matrix columns
                    Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                        [&] (const unsigned j) { // nested lambda
                            tmp[j] *= row[j];
                        });
                }
            }

            // accumulate contribution to row k of result
            Kokkos::parallel_for(Kokkos::ThreadVectorRange(team,R),
                [&] (const unsigned j) { // nested lambda
                    Kokkos::atomic_add(&V.entry(k,j), tmp[j]);
                });
        });
    }
}

Fig. 5: More advanced Kokkos implementation of MTTKRP that is row-based, using team-parallelism and scratch pad memory. Note that the nested lambdas for vector parallelism do not use KOKKOS_LAMBDA since they are captured and executed within the same execution space (host or device) that the outer lambda is executed in.

addressed by tiling the nested parallel loops over $R$, using some small, fixed tile size to limit the amount of shared memory allocated by each team. This requires modifying the kernel to include an additional outer loop over factor matrix tiles, and additional logic to handle the remainder when $R$ is not divisible by the tile size.

The downside to this approach is that the tile size needs to be fairly small due to
the small shared memory sizes available on contemporary GPUs. For example, with a team size of 16, the algorithm is limited to a maximum tile size of 32 (in double precision) to have 16 CUDA blocks active on each GPU SM (for 100% occupancy) with 64 KB of available shared memory \((16 \times 16 \times 32 \times 8 \text{ B} = 64 \text{ KB})\). This limits reuse of tensor nonzero indices since they must be re-read for each tile. Because the temporary buffer is allocated dynamically on CPU/Phi platforms, it is unlikely the compiler will store these values in vector registers. Furthermore, the use of lambda expressions for the innermost for-loops can make it difficult for the compiler to vectorize.

5. MTTKRP using Kokkos with compile-time polymorphic arrays. To address the problem of having too much shared memory usage, we propose a more performant and arguably simpler approach to portability, which we refer to as compile-time polymorphic arrays. The idea is inspired by optimized CPU- and GPU-specific versions. On a GPU, if we tile the matrix columns so that each group of columns is a small multiple of the vector size and apply vector/warp parallelism across those columns, then each CUDA thread within the vector/warp only processes a few factor matrix columns. Then instead of storing the temporary buffer in a single shared memory array accessible to all threads within the vector, each thread can instead store its portion of the temporary buffer in thread-local register variables. Furthermore, on a CPU-like architecture where vector parallelism is implemented through vector instructions local to each thread, the temporary buffer can be stored in a simple, thread-private stack array which the compiler is much more likely to replace with vector registers.

Both approaches can be supported by introducing an array wrapper class that has different implementations for each architecture. In GenTen we call this class TinyVec, and the default implementation is shown in Figure 6 with a specialization for CUDA in Figure 7. The class has a number of template parameters, including the execution space in which the array will be used (Space), the total length of the array to store (Length), the total size of the portion of the array to be used (Size), the vector/warp dimension for GPU architectures (WarpDim), and a template parameter for disambiguating partial specializations (Enabled). The class stores a statically-sized, thread-private array (aligned to 64 bytes via alignas to improve vectorization performance) and overloads various arithmetic operations mapped across the array. The length of the array allocated by each GPU thread within a vector/warp is determined by dividing Length by WarpDim, so that if the length is 64 and the vector dimension is 32, each thread will allocate a local array of length 2.\(^4\) On CPU/Phi architectures, WarpDim will be set to one.

A second dimension is supplied to TinyVec through Size which determines what portion of the array to actually use in its overloaded operators. In general, Size can be any integer less than or equal to Length (divisible by WarpDim), but in its use in GenTen, Size equals Length or is zero. The size to use is stored through the integral_nonzero_constant helper class. This class has two specializations depending on whether the second template parameter (Size/WarpDim) is zero or nonzero. If it is nonzero then sz.value will be a compile-time constant enum equal to its second template parameter. However if it is zero, sz.value is a data member given by the type of its first template parameter (unsigned) which is initialized through its constructor. In TinyVec, sz.value is used as the loop bound in each overloaded operator, and therefore this technique allows the same implementation support loop

\(^4\)It is assumed that Length and Size are evenly divisible by WarpDim.
template <typename Space, unsigned Length, unsigned Size, unsigned WarpDim, typename Enabled = void>
struct TinyVec {
    Kokkos::Impl::integral_nonzero_constant< unsigned, Size/WarpDim > sz;
    alignas(64) double v[ Length / WarpDim ];

    KOKKOS_INLINE_FUNCTION TinyVec(const unsigned size, const double x) :
        #ifdef __CUDA_ARCH__
        sz( (size+WarpDim-1-threadIdx.x) / WarpDim )
        #else
        sz(size)
        #endif
    {
        for (unsigned i =0; i<sz.value; ++i) v[i] = x;
    }

    KOKKOS_INLINE_FUNCTION void store_plus(double* x) const {
        #ifdef __CUDA_ARCH__
        for (unsigned i =0; i<sz.value; ++i) x[i*WarpDim+threadIdx.x] += v[i];
        #else
        for (unsigned i =0; i<sz.value; ++i) x[i] += v[i];
        #endif
    }

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

    KOKKOS_INLINE_FUNCTION TinyVec & operator *=( const double x);
    KOKKOS_INLINE_FUNCTION TinyVec & operator *=( const double * x);
};

Fig. 6: Default implementation of thread-local compile-time polymorphic array wrapper class. The implementations of several functions, which are similar to the implementations shown, are suppressed for brevity.

bounds determined at compile and run times. For its use in GenTen, the factor matrix column loop is blocked by some compile-time fixed value, which determines $\text{Length}$ and $\text{Size}$. This results in a factor matrix block loop, where each inner iteration of this loop process $\text{Length} == \text{Size}$ columns using the first specialization of $\text{integral\_nonzero\_constant}$. However if the number of factor matrix columns ($R$) is not divisible by the blocking size, the last iteration of this loop processes a run-time determined number of factor matrix columns by setting the second template parameter to zero.

The implementation in Figure 6 will work on CPU, Phi, and CUDA GPU architectures since each member function that involves reading or writing to memory (such as $\text{store\_plus}$ or the overload of $\text{*=}$ with a pointer argument) has a separate implementation for CUDA to use the $\text{WarpDim}$ template parameter for obtaining the properly strided loads/stores.

Since the class stores its array as a stack array, it is likely the compiler can replace this array with (vector) registers on CPU/Phi platforms. The same is true on NVIDIA GPU architectures, except in the dynamically sized case when $\text{Size}$ is zero. Due to limitations on how NVIDIA GPUs allocate registers, the stack array will be stored in what is called “local” memory when loops indexing into this memory do not have a compile-time known trip count. Unfortunately local memory accesses have
#if defined(KOKKOS_HAVE_CUDA) && defined(_CUDA_ARCH_)

## TinyVec Template

```cpp
template <unsigned Length, unsigned Size, unsigned WarpDim, 
          typename std::enable_if<Length/WarpDim == 4>::type > {
  Kokkos::Impl::integral_nonzero_constant<unsigned_type, Size/WarpDim> sz;
  double v0, v1, v2, v3;

  __device__ inline TinyVec(const unsigned size, const double x): sz((size + WarpDim - 1 - threadIdx.x) / WarpDim) {
    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[WarpDim + threadIdx.x] += v1;
    if (sz.value > 2) x[2*WarpDim + threadIdx.x] += v2;
    if (sz.value > 3) x[3*WarpDim + 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);
};
#endif
```

Fig. 7: Specialization of thread-local compile-time polymorphic array wrapper class for CUDA when $\text{Length}/\text{WarpDim} = 4$. The specialization uses typical class template specialization techniques in conjunction with Substitution Failure Is Not An Error (SFINAE) [27] using `std::enable_if` for the `Enabled` template parameter to make the specialization available only when $\text{Length}/\text{WarpDim} = 4$.

The implementation of MTTKRP for a single block of tensor nonzeros and factor matrix columns using the polymorphic array `TinyVec` is displayed in Figure 8. The factor matrix tile size is determined by the `FacBlockSize` template parameter, which becomes the `Length` template parameter of `TinyVec`. The template parameter `Nj`
template <typename Space, unsigned FacBlockSize, unsigned Nj, unsigned VectorSize>
KOKKOS_INLINE_FUNCTION void mttkrp_kernel_block(
    const Sptensor<Space>& X, const Ktensor<Space>& M, const unsigned n,
    const FacMatrix<Space>& V, const TeamMember& team, const unsigned EntryBlockSize,
    const unsigned j, const unsigned nj)
{
    typedef TinyVec<Space, FacBlockSize, Nj, VectorSize> TV;
    const size_t nnz = X.nnz();
    const unsigned d = M.ndims();
    // Scatter threads across tensor nonzeros on the GPU with a large stride
    size_t offset, stride;
    if (Genten::is_cuda_space<Space>::value) {
        offset = team.league_rank()*team.team_size()+team.team_rank();
        stride = team.league_size()*team.team_size();
    } else {
        offset = (team.league_rank()*team.team_size()+team.team_rank())*EntryBlockSize;
        stride = 1;
    }
    for (unsigned ii=0; ii<EntryBlockSize; ++ii) {
        const size_t i = offset + ii*stride;
        if (i >= nnz) return;
        const size_t k = X.subscript(i,n);
        const double x_val = X.value(i);
        TV tmp(nj, x_val);
        tmp *= &M.weights(j);
        for (unsigned m=0; m<d; ++m) {
            if (m != n)
                tmp *= &M[m].entry(X.subscript(i,m),j);
        }
        tmp.atomic_store_plus(&V.entry(k,j));
    }
}

Fig. 8: MTTKRP kernel implementation for a single block of tensor nonzeros and factor matrix columns.

and function argument nj determine the size of TinyVec that is used. For the inner iterations of the factor matrix block loop, this function will be called with Nj equal to FacBlockSize, resulting in the static size case described above. But for the final iteration, Nj will be zero and nj will determine the TinyVec size to handle the remainder. The starting offset of the factor matrix columns is given by j. Thread team information is passed through the team argument, which is used to determine which tensor nonzeros are processed by each team. Each team member process a number of tensor nonzeros as determined by EntryBlockSize, which in GenTen is currently always 128. However different mechanisms are employed to map tensor nonzeros to each team on the GPU versus CPU/Phi. In the CPU/Phi case, each team member processes a contiguous block of nonzeros as they are stored in the tensor, but on the GPU, a large stride is used to spread team members out across tensor nonzeros (similar to the approach in the ParTI software [15, 16]). This improves performance considerably when tensor entries are sorted lexicographically by preventing a large number of team members from writing to the same row of V simultaneously.

The use of TinyVec simplifies the MTTKRP block implementation by removing the explicit loop over factor matrix columns. No nested parallel-for’s and lambda expressions are needed, yet (as will be shown) good performance is obtained due to good vectorization on the CPU/Phi and storing the factor matrix accumulation buffer.
Table 1: Vector and factor block sizes for different numbers of components ($R$)

| 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 4 8 8 8 8 16 16 32 32 | Fac. Bl. Size | 1 2 4 4 8 8 16 24 32 48 64 96 128 |

in registers on the GPU. The MTTKRP code is essentially agnostic to the underlying architecture (apart from the tensor nonzero mapping) since TinyVec hides the architecture-specific vector parallelism implementation. Also, since no shared memory is used, a much larger tile size can be used while still maintaining high-occupancy. Also, by using overloaded operators that mimic standard scalar operations, the implementation is very similar to one where a single factor matrix column is processed at a time.

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 tensor nonzeros) 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 1, 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 choices result in at most four factor matrix columns processed by each thread within a warp (and thus Length/WarpDim is at most four). The code for computing these sizes and calling mttkrp_kernel_block is discussed in Appendix A.

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 compare-and-swap.

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 [25]. 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 coordinate (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 they are stored in memory. This requires storing a two-dimensional permutation array of the same size and shape as the tensor coordinates, and therefore a little less than double the amount of memory is required to store the tensor. The downside of this approach is 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; see Figure 9. The same logic is used for determining the vector and factor matrix tile sizes as shown in Table 1 and Appendix A, and only an overload of `mttkrp_kernel_block()` is required for the new sparse tensor class `Sptensor_perm` which stores an additional permutation array (accessed by its `getPerm()` member function). 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 (non-atomic) write. Determining when and what kind of write should happen results in most of the changes to the kernel implementation. 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. GenTen can also use parallel sorting routines provided by Thrust\(^5\), which is included in the NVIDIA CUDA toolkit for GPU architectures as well as the Intel Parallel Stable Sort\(^6\) for OpenMP-based architectures.

7. Numerical results. We investigate the parallel performance of our Kokkos-based CP-ALS implementation on a variety of computing architectures, as summarized in Table 2. Our results are generated with the publicly available GenTen library with compilers listed in Table 2 and architecture/compiler-specific optimization flags chosen by Kokkos.

7.1. Artificial data and scalability studies. We first consider running the full CP-ALS algorithm on a synthetic three-dimensional tensor of size $30K \times 40K \times 50K$ with $10M$ nonzeros placed randomly throughout the tensor. For each architecture, Figure 10 displays the total run time for 10 iterations of CP-ALS with $R = 128$ factor components. For the Haswell and KNL architectures, Figure 10a is generated by varying 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

\(^5\)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
\(^7\)On both architectures these results were generated with `OMP_PROC_BIND=close` and `OMP_PLACES=threads` OpenMP thread-binding environment variables set.
template <typename Space, unsigned FacBlockSize, unsigned Nj, unsigned VectorSize>
KOKKOS_INLINE_FUNCTION void mttkrp_kernel_block(
    const Sptensor_perm<Space>& X, const Ktensor<Space>& M, const unsigned n,
    const FacMatrix<Space>& V, const TeamMember& team, const unsigned EntryBlockSize,
    const unsigned j, const unsigned nj)
{
    typedef TinyVec<Space, FacBlockSize, Nj, VectorSize> TV;
    const size_t nnz = X.nnz();
    const unsigned d = M.ndims();
    const size_t offset =
        (team.league_rank() * team.team_size() + team.team_rank()) * EntryBlockSize;

    const size_t invalid_row = size_t(-1);
    size_t row_prev, row, first_row, p = invalid_row;
    double x_val = 0.0;
    TV val(nj, 0.0), tmp(nj, 0.0);

    for (unsigned ii = 0; ii < EntryBlockSize; ++ii) {
        const size_t i = offset + ii;
        if (i >= nnz)
            row = invalid_row;
        else {
            p = X.getPerm(i, n);
            x_val = X.value(p);
            row = X.subscript(p, n);
        }
        if (ii == 0) first_row = row;
        if (row != row_prev) { // If we got a different row index, add in result
            if (row_prev != invalid_row) {
                if (row_prev == first_row) // Only need atomics for first/last row
                    val.atomic_store_plus(&V.entry(row_prev, j));
                else
                    val.store_plus(&V.entry(row_prev, j));
                val = 0.0;
            }
            row_prev = row;
        }
        if (row != invalid_row) {
            tmp.load(&u.weights(j));
            tmp *= x_val;
            for (unsigned m = 0; m < d; ++m) {
                if (m != n)
                    tmp *= &(M[m].entry(X.subscript(p, m), j));
            }
            val += tmp;
        }
    }

    if (row != invalid_row) // Sum in last row
        val.atomic_store_plus(&V.entry(row, j));
}

Fig. 9: Permuted MTTKRP kernel implementation for a single tensor nonzero and factor matrix block.

number of threads used. For the GPU architectures, it is not possible to vary the
number of threads and so Figure 10b displays the total run-time for all four architectures at full machine capacity. These plots demonstrate good thread scalability on the Haswell and KNL architectures, and similar performance for the Haswell, KNL, and K80 architectures at full machine capacity. The Pascal P100 GPU is substantially faster than the other three. As compared to single Haswell core, we see about a 20-fold speedup on Haswell and KNL, a 34-fold speedup on the K80, and a 140-fold speedup on the P100.
Table 2: 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.

| Type | 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 17.1 |
| GPU  | K80 NVIDIA Kepler K80 GPU | CUDA | NVCC 8.0 |
| GPU  | P100 NVIDIA Pascal P100 GPU | CUDA | NVCC 8.0 |

Fig. 10: Comparison of performance on different architectures in terms of total run time for 10 iterations of CP-ALS with $R = 128$ on a tensor of size $30K \times 40K \times 50K$ with 10M nonzeros. (a) CP-ALS time as a function of the number of threads used on Haswell CPU and KNL platforms, normalized by the maximum available (the number of threads used cannot be varied on GPU platforms). (b) CP-ALS time on all platforms using the maximum threads available, decomposing the time used by MTTKRP and all other kernels.

On the same data tensor, we compare the original and permuted sparse tensor data structures in Figure 11, measured as the MTTKRP run time over averaged over 10 iterations of CP-ALS. Substantial reductions in computational cost are observed for the Haswell, KNL, and K80 architectures. Hence, atomic-write throughput is a substantial bottleneck in MTTKRP performance and is dramatically improved with the permuted data structure/algorithm. We see less 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.

To understand how our performance compares to what we might expect, we do some simple analysis. If we assume all tensor and factor components values are read
from main memory with no caching and no thread contention for atomic-writes, then the expected MTTKRP performance is

\[
\text{GFLOP/s} = \frac{R(d+1)}{(dR+3)s_r + ds_o} B \approx \frac{B}{s_r}
\]

where \( B \) is the bandwidth (in GB/s) to/from main memory, \( s_r \) is the size (in bytes) of the floating-point type and \( s_o \) is the size of the ordinal type. To determine the bandwidth, we used the STREAM benchmark [19]. The measured bandwidth and resulting floating-point throughput estimated from (2) is given in Table 3. The measured throughput for both the original and permuted algorithms on all four architectures for \( R \in [8, 256] \) with a stride of 8 is shown in Figure 12, with the green dotted line indicating the throughput derived from the estimated bandwidth. For all four architectures, the throughput of the permuted approach is closer to the estimated throughput since the cost of the atomic writes is substantially reduced, making the estimate a better model of the actual MTTKRP calculation. The performance can be better than the estimate when we get 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 measured throughput only approaches the estimate 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 see the throughput 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 1.

7.2. Real-world data and comparisons to SPLATT. We study the MTTKRP performance on several tensors available from the Formidable Repository of Open Sparse Tensor and Tools (FROSTT) [22]. We selected tensors to be as large

Fig. 11: Comparison of MTTKRP performance for original and permuted sparse tensor data structures, running on different architectures. We report the total MTTKRP time for 10 iterations of CP-ALS with \( R = 128 \) on a tensor of size 30K \( \times \) 40K \( \times \) 50K with 10M nonzeros. (a) MTTKRP time as a function of the fraction of the machine used on Haswell and KNL platforms. (b) MTTKRP time at full machine capacity for all architectures.
Table 3: Measured bandwidth and estimated floating-point throughput for each architecture based on 64 bit floating-point tensor values.

| Architecture | Bandwidth (GB/s) | Estimated Throughput (GFLOP/s) |
|--------------|------------------|-------------------------------|
| HSW          | 101              | 12.6                          |
| KNL          | 300              | 37.5                          |
| K80          | 150              | 18.8                          |
| P100         | 531              | 66.4                          |

Table 4: Real-world tensor data sets from FROSTT.

| 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      |
| Delicious| 4 | 532K × 17M × 2.5M × 1.4K | 140M     |

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 Table 4.

We compare the original and permuted version of our algorithm as well as the leading state-of-the-art method, SPLATT’s CSF-based algorithms [23], using mutexes (SPLATT-M) and tiling without mutexes (SPLATT-T) and both using the default of two CSF modes. SPLATT does not have a GPU implementation, so we only include it for the two OpenMP-based architectures (Haswell and KNL). Table 5 shows the results based on the total MTTKRP time for 10 CP-ALS iterations for the original and permuted algorithms with $R = 16$ factor components.

If we compare just the original and permuted approaches, the permuted approach is oftentimes 1-2 orders of magnitude faster. The more pronounced difference for more realistic data tensors is due to higher atomic contention. This is particularly pronounced on the K80 architecture due to its lack of double-precision atomic instructions, where several orders of magnitude speed-up are observed. However, the cost of storing the permutations exhausted the global memory on the K80 for the largest tensor (with 140M nonzeros). The main exception to the permuted approach being better is the P100, which does not suffer from atomic contention. As a result, there is little advantage to the permuted approach and it is sometimes even slower.

If we compare to SPLATT on HSW, our permuted approach is 50% faster in one case and never more than four times slower. On KNL, our approach is more than three times faster of the largest tensor, and never worse than three times slower. Overall, we claim that the performance of our code is comparable to that of SPLATT (i.e., same order of magnitude) while having the advantage of being portable.

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 permutation-based MTTKRP algorithm, scaled by the average (permuted) CP-ALS iteration time, 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 a relatively small but not inconsequential cost.

8. Conclusions. In this paper we describe the development and implementation of MTTKRP for sparse tensors on emerging computer architectures, including multicore CPUs, manycore Intel Xeon Phis, and NVIDIA GPUs. Portability is facilitated by using Kokkos, which provides data structures and abstractions so that a single implementation achieves good performance on these architectures. Our resulting implementation of MTTKRP as well as the full CP-ALS method is available in the open-source software package GenTen. We introduced a novel extension of the Kokkos framework based on the use of polymorphic data arrays that allows temporary data to be stored in registers, yet whose length is parameterized by a compile-time constant. We also described a new sparse tensor data format that adds a sorted index with respect to each mode to the traditional coordinate-based storage format (requiring approximately double the space of the standard format); this addition can
Table 5: Total MTTKRP time (in seconds) for 10 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 before the CP-ALS algorithm could complete for the Delicious tensor on the K80 architecture.

| Arch. | Method | LBNL | Uber | Enron | VAST | NELL2 | Delicious |
|-------|--------|------|------|-------|------|-------|-----------|
| HSW   | Orig   | 14.7 | 8.6  | 41.0  | 281.9| 42.8  | 56.9      |
|       | Perm   | 0.4  | 0.3  | 4.4   | 6.1  | 5.0   | 22.5      |
|       | SPLATT-M| 0.8 | 0.2  | 1.1   | 26.0 | 1.3   | 23.5      |
|       | SPLATT-T| 0.5 | 1.6  | 10.5  | 3.8  | 1.4   | 71.9      |
| KNL   | Orig   | 41.5 | 9.4  | 109.1 | 1849.5| 82.0  | 79.9      |
|       | Perm   | 0.4  | 0.2  | 2.8   | 16.4 | 3.1   | 13.3      |
|       | SPLATT-M| 2.0 | 1.2  | 2.4   | 150.1| 8.4   | 46.7      |
|       | SPLATT-T| 1.0 | 5.1  | 40.7  | 6.5  | 14.3  | 216.8     |
| K80   | Orig   | 71.9 | 5.5  | 168.1 | 1998.1| 54.5  | 32.7      |
|       | Perm   | 0.4  | 0.5  | 9.3   | 7.9  | 11.7  | –         |
| P100  | Orig   | 0.4  | 0.2  | 2.0   | 4.2  | 1.6   | 4.1       |
|       | Perm   | 0.1  | 0.1  | 1.9   | 1.6  | 1.9   | 11.3      |

Table 6: Sorting cost for the permutation-based MTTKRP approach scaled by the average (permuted) CP-ALS iteration time with $R = 16$ for each data tensor and computational architecture.

| Architecture | LBNL | Uber | Enron | VAST | NELL2 | Delicious |
|--------------|------|------|-------|------|-------|-----------|
| HSW          | 0.7  | 4.8  | 11.3  | 5.3  | 8.9   | 4.0       |
| KNL          | 1.0  | 5.6  | 8.2   | 1.2  | 7.7   | 4.8       |
| K80          | 1.1  | 3.4  | 3.6   | 2.7  | 2.8   | –         |
| P100         | 0.5  | 1.3  | 4.1   | 3.5  | 4.6   | 2.5       |

be used to reduce atomic contention in the CP-ALS algorithm, leading to substantially improved performance. The performance of GenTen was studied on a variety of contemporary architectures and we 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 on CPU and KNL platforms using several realistic data tensors from FROSTT, demonstrating comparable or better performance for the permutation-based MTTKRP algorithm.

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). 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). Finally we are 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 [24]). 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. This suggests a heuristic approach that chooses between the atomic and reduction approaches based on these sizes, or possibly a hybrid approach where threads are divided into groups and each group has a private copy of the factor matrix but threads within the group write to their copy atomically.

Appendix A. MTTKRP kernel implementation. The GenTen code that determines the factor matrix block and vector sizes displayed in Table 1 is shown in Figure 13. First, the top-level function \texttt{mttkrp()} in Figure 13 chooses a vector size between 1 and 32, depending on the size of \( R \). It then calls \texttt{mttkrp\_kernel()} with the supplied vector size. This function then determines an appropriate tile size for \( R \), between one and four times the vector size so that in the CUDA case, each CUDA thread processes between one and four factor matrix columns within one invocation of the lowest level kernel. This then calls \texttt{mttkrp\_kernel\_impl()} which implements the parallel-for loop over tensor nonzeros (these functions are templated on the sparse tensor type to allow them to be used with multiple tensor data formats, including the permutation approach). This uses a team-based execution policy using the supplied vector size, and a team size of one on the CPU/Phi and \( 128/\text{VectorSize} \) on the GPU. Each team member also processes 128 tensor entries, and so the league size is computed accordingly. The body of the parallel-for then loops over factor matrix columns in tiles of size \( \text{FacBlockSize} \). The number of columns processed is \( \text{FacBlockSize} \) for each iteration, except the last iteration when \( R \) is not evenly divided by \( \text{FacBlockSize} \). Each iteration of the loop calls \texttt{mttkrp\_kernel\_block()} which implements the MTTKRP algorithm for a single block of tensor entries and factor matrix columns as shown in Figure 8. This function then employs \texttt{TinyVec} to loop over factor matrix columns within the block. Notice the length of the \texttt{TinyVec} array is always \( \text{FacBlockSize} \), but the size of the array that is used is either \( \text{FacBlockSize} \) (statically sized case) or \( R - j \) (dynamically sized case for the remainder) as determined by whether the \( \&j \) template parameter is nonzero.

Appendix B. CP-ALS kernels beyond MTTKRP. The Kokkos implementations of other CP-ALS kernels such as the Gramian, norm, scale and inner product kernels are quite similar to the kernels described above, and therefore aren’t described in detail. The only difference is they deal only with factor matrices, and therefore the number of rows of the factor matrix is used for the league dimension instead of the number of tensor nonzeros. Currently, GenTen only provides a serial fall-back implementation of the solve kernel, and relies on optimized implementations for each architecture. For Intel CPU/Phi architectures, we use \texttt{gesv} from the Intel MKL library. For NVIDIA GPUs we use \texttt{getrf/getrs} from the cuSolver library provided as part of the NVIDIA CUDA toolkit. Similarly, GenTen can use \texttt{gemm} from Intel MKL or cuBLAS for the Gramian instead of the Kokkos implementation provided.

Acknowledgments. We graciously acknowledge funding support from the Lab Directed Research and Development (LDRD) program and Sandia National Laboratories.

REFERENCES

[1] B. W. Bader and T. G. Kolda, Algorithm 862: MATLAB tensor classes for fast algorithm prototyping, ACM Transactions on Mathematical Software, 32 (2006), pp. 635–653, \url{https://doi.org/10.1145/1186785.1186794}. 
template <typename SparseTensor>
void mttkrp(const SparseTensor& X,
        const Ktensor<typename SparseTensor::space>& M,
        const size_t n,
        const FacMatrix<typename SparseTensor::space>& V) {
    const size_t R = M.ncomponents();
    v = double(0.0);
    if (R >= 96)
        mttkrp_kernel<32>(X,M,n,V);
    else if (R >= 48)
        mttkrp_kernel<16>(X,M,n,V);
    else if (R >= 8)
        mttkrp_kernel<8>(X,M,n,V);
    else if (R >= 4)
        mttkrp_kernel<4>(X,M,n,V);
    else if (R >= 2)
        mttkrp_kernel<2>(X,M,n,V);
    else
        mttkrp_kernel<1>(X,M,n,V);
}

template <unsigned VS, typename SparseTensor>
void mttkrp_kernel(const SparseTensor& X,
            const Ktensor<typename SparseTensor::space>& M,
            const size_t n,
            const FacMatrix<typename SparseTensor::space>& V) {
    static const unsigned VS4 = 4*VS;
    static const unsigned VS3 = 3*VS;
    static const unsigned VS2 = 2*VS;
    static const unsigned VS1 = 1*VS;
    const unsigned R = M.ncomponents();
    if (R > VS3)
        mttkrp_kernel_impl<VS,VS4>(X,M,n,V);
    else if (R > VS2)
        mttkrp_kernel_impl<VS,VS3>(X,M,n,V);
    else if (R > VS1)
        mttkrp_kernel_impl<VS,VS2>(X,M,n,V);
    else
        mttkrp_kernel_impl<VS,VS1>(X,M,n,V);
}

template <unsigned VectorSize, unsigned FacBlockSize, typename SparseTensor>
void mttkrp_kernel_impl(const SparseTensor& X,
            const Ktensor<typename SparseTensor::space>& M,
            const size_t n,
            const FacMatrix<typename SparseTensor::space>& V) {
    typedef typename SparseTensor::space Space;
    typedef Kokkos::TeamPolicy<Space> Policy;
    const unsigned R = M.ncomponents();
    const unsigned TeamSize = Genten::is_cuda_space<Space>::value ? 128/VectorSize : 1;
    const unsigned EntriesPerTeam = 128;
    const unsigned EntriesPerTeam = (nnz+EntriesPerTeam-1)/EntriesPerTeam;
    Policy policy(LeagueSize ,TeamSize ,VectorSize);
    Kokkos::parallel_for(policy,
            KOKKOS_LAMBDA(typevalue Policy::member_type team) {
            for (unsigned j=0; j<R; j+=FacBlockSize) {
                if (j+FacBlockSize <= R)
                    mttkrp_kernel_block<FacBlockSize,FacBlockSize,VectorSize>(
                        X, M, n, V, team, EntryBlockSize, j, FacBlockSize);
                else
                    mttkrp_kernel_block<FacBlockSize,0,VectorSize>(
                        X, M, n, V, team, EntryBlockSize, j, R-j);
            }
            });
}

Fig. 13: MTTKRP implementation including logic to determine vector size and factor matrix block size from $R$. 
[2] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," SIAM Journal on Scientific Computing, 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, New York, NY, USA, 2009, ACM, pp. 18:1–18:11, 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, PMLR, June 2016, pp. 698–728, http://proceedings.mlr.press/v49/chen16.html.

[7] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, Manycore performance-portability: Kokkos multidimensional array library, Scientific Programming, 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, Journal of Parallel and Distributed Computing, 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, http://www.sciencedirect.com/science/article/pii/S0950705116000472.

[10] E. Gursul, R. Pasbicha, and E. E. Papalexakis, Samutaten: Sampling-based batch incremental tensor decomposition, https://arxiv.org/abs/1709.00668v1.

[11] R. A. Harshman, Foundations of the PARAFAC procedure: Models and conditions for an “explanatory” multi-modal factor analysis, UCLA working papers in phonetics, 16 (1970), pp. 1–84. Available at http://www.psychology.uwo.ca/faculty/harshman/wpppfac0.pdf.

[12] M. Janzamin, H. Sedghi, and A. Anandkumar, "Beating the perils of non-convexity: Guaranteed training of neural networks using tensor methods," https://arxiv.org/abs/1506.08473v3.

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

[14] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, Model-Driven Sparse CP Decomposition for Higher-Order Tensors, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 2017, pp. 1048–1057, https://doi.org/10.1109/IPDPS.2017.80.

[15] J. Li, Y. Ma, and R. Vuduc, ParTI!: A parallel tensor infrastructure, 2017, https://github.com/lpggarage/ParTI.

[16] J. Li, Y. Ma, C. Yan, and R. Vuduc, Optimizing sparse tensor times matrix on multi-core and many-core architectures, in Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, IA3 ’16, Piscataway, NJ, USA, 2016, IEEE Press, pp. 26–33, https://doi.org/10.1109/IA3.2016.10.

[17] J. Li, J. Sun, and R. Vuduc, HiCOO: Hierarchical Storage of Sparse Tensors, in ACM/IEEE International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC18), 2018.

[18] B. Liu, C. Wen, A. D. Sarwate, and M. Mehdi Dehnavi, A Unified Optimization Approach for Sparse Tensor Operations on GPUs. ArXiv e-prints, 2017, https://arxiv.org/abs/1705.09905.

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

[20] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, Tensorizing neural networks, CoRR, abs/1509.06569 (2015), http://arxiv.org/abs/1509.06569.

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

[22] 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/.

[23] 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,
[24] S. Smith, J. Park, and G. Karypis, *Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory*, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 2017, pp. 1058–1067, https://doi.org/10.1109/IPDPS.2017.84.

[25] 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, May 2015, pp. 61–70, https://doi.org/10.1109/IPDPS.2015.27.

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

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

[28] 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, New York, NY, USA, 2015, ACM, pp. 1265–1274, https://doi.org/10.1145/2783258.2783395, http://doi.acm.org/10.1145/2783258.2783395.