Performance Portability of Sparse Tensor Decomposition Operations

S. Isaac Geronimo Anderson
University of Oregon
igeroni3@uoregon.edu

Jee Choi
University of Oregon
jeec@uoregon.edu

***Abstract*—We leverage the Kokkos library to study the performance portability of parallel sparse tensor decompositions on CPU and GPU architectures. Real-world multi-way data can be represented using a multi-dimensional array, or *tensor*, and tensor rank decomposition can reveal latent information within data. Tensors storing real-world data are often large and sparse, necessitating space-efficient storage and time-efficient parallel algorithms. CANDECOMP/PARAFAC via Alternating Poisson Regression Multiplicative Update (CP-APR MU) is a memory bandwidth–bound algorithm which calculates tensor rank decomposition for count data, and which is composed of simple array operations. We compare the performance of Kokkos implementations of three kinds of kernels (simple array operations, MTTKRP, and CP-APR MU) to platform-specific implementations on CPUs and GPUs.**

Our result shows that *with a single implementation* Kokkos can deliver performance comparable to hand-tuned code for simple array operations that make up tensor decomposition kernels on a wide range of CPU and GPU systems, and *superior* performance for the MTTKRP kernel on CPUs, but exhibits comparable to lower performance in the case of the CP-APR MU kernel on CPU systems.

1. Introduction

Tensors are the higher-order generalization of matrices, and tensor decompositions (or factorizations) provide a useful tool for analyzing latent relationships in multi-way data [1]. Many real-world data analysis applications in various areas—e.g., in healthcare, cybersecurity, social networks, and more—give rise to multi-way data that can be naturally represented by sparse tensors.

Performance–portability analyses for *parallel tensor decomposition* have been conducted at the application level with software such as SparTen [2] and GenTen [1], which respectively decompose sparse tensors via the CANDECOMP/PARAFAC Alternating Poisson Regression (CP-APR) and CANDECOMP/PARAFAC Alternating Least Squares (CP-ALS) algorithms. Here we explore the performance portability of parallel tensor decomposition at the level of *essential array operations* by implementing a set of proxy benchmark kernels using the Kokkos C++ performance-portable library [3]. With the emergence of drastically different parallel architectures, performance portability is critical in achieving optimal productivity on heterogeneous computing systems.

CP-APR and CP-ALS are algorithms for computing the canonical polyadic decomposition model, which approximates a tensor as a sum of R rank-one tensors [1]. CP-APR is used similarly to CP-ALS with the distinction that CP-APR is more effective at modeling data with a Poisson distribution, while CP-ALS models data with a Gaussian distribution [4]. Two key performance bottlenecks in tensor decomposition are the matrixized tensor times Khatri-Rao product (MTTKRP) used in CP-ALS, and the multiplicative update (MU) operation used in the CP-APR MU algorithm. The MTTKRP and MU kernels are composed from simple array operations, and these array operations form the basis of this study. We start by evaluating parallel performance with simple array operations from the STREAM benchmark [5], and then extend the evaluation to the MTTKRP kernel, and finish by evaluating the multiplicative update (MU) kernel.

Contributions. We make three key contributions towards performance portable sparse tensor decomposition operations:

1) *Augmentation* of existing STREAM and MTTKRP implementations using Kokkos for portability.
2) Evaluation of Kokkos and manually-tuned benchmarks on several CPU and GPU architectures.
3) Analysis of the performance portability of tensor decomposition algorithms on multiple architectures.

These contributions collectively provide insight into the performance portability relationships between fundamental array operations and sparse tensor decomposition operations on a variety of systems and processor organizations, and they provide a case study of Kokkos as a means for leveraging these systems.

2. Background

This section summarizes tensors (§2.1) and tensor rank decomposition (§2.2); We direct the reader to Kolda and Bader [6] for their essential and detailed discussion of tensors, tensor decompositions, and related algorithms.

2.1. Tensors and Sparse Tensors

Real-world multi-way data can be represented using a multi-dimensional array, or *tensorial array*, commonly referred to simply as a *tensor* [6]. For example, consider a three-dimensional tensor $\mathcal{S}$ whose dimensions are user, song, and year; this tensor can store entries which are the number

of minutes a particular user has spent listening to a particular song in a particular calendar year. Formally, tensors are N-dimensional arrays, where each array element has a corresponding N-tuple index $\mathbf{i} = (i_1, i_2, \ldots, i_N)$. Each index coordinate i_k fixes an element's location along the k^{th} dimension, with $k \in \{1, 2, \ldots, N\}$ and $i_k \in \{1, 2, \ldots, I_k\}$. Low-dimensional tensors include vectors, where $N = 1$, and matrices, where $N = 2$. Thus a length I_1 vector has I_1 indexed elements, and an $I_1 \times I_2$ matrix has I_1 rows and I_2 columns for indexing its $I_1 I_2$ elements. In general, an N-dimensional (or N-*mode*) tensor with dimensions $I_1 \times \cdots \times I_n$ represents $\prod_{k=1}^{N} I_k$ distinct coordinates each with a corresponding element. This paper uses additional definitions and notation as listed in Table 1.

TABLE 1: Notation used in this paper

.

Notation	Definition
i, i_k	Scalars (italic letters)
$\mathbf{a}, \mathbf{i}$	Vectors and tuples (bold lower-case letters)
$\mathbf{B}, \mathbf{\Phi}$	Matrices (bold upper-case letters)
$\mathcal{X}, \mathcal{M}$	Sparse tensors and tensor models (script letters)
R	Rank, the desired number of model components
nnz	Number of non-zero values in a sparse tensor
$\mathcal{X}_i$	The i-th non-zero value in a sparse tensor $\mathcal{X}$
$\texttt{coord}_{n,\mathcal{X},i}$	The n-coordinate for $\mathcal{X}_i$
λ	Weight vector
$\mathbf{\Lambda}$	Diagonal weight matrix ($\text{diag}(\lambda)$)
$\mathbf{A}^{(n)}$	Mode-n model factor matrix
$\mathbf{B}$	Intermediate representation of $\mathbf{A}^{(n)}$
$\mathbf{\Pi}$	Intermediate calculation matrix
$\mathbf{\Phi}$	Intermediate calculation matrix
$\mathbf{X}_{(n)}$	Mode-n flattening of $\mathcal{X}$
ϵ	Minimum divisor to prevent divide-by-zero
τ	Convergence error tolerance
κ_{min}	Inadmissible zero minimum
κ_{adj}	Inadmissible zero adjustment value
$\mathbf{1}$	All-ones vector
$\oslash$	Element-wise division
$\circ$	Element-wise multiplication
$\max^{\circ}$	Element-wise maximum

Sparse tensors are tensors where most elements are zero, and are analogous to sparse matrices. Real-world tensors are often large and sparse. For example, consider a three-dimensional tensor $\mathcal{S}$ whose dimensions are user, song, and year. If there are many songs and many users, it is likely that not every user has listened to every song in every year, which means that many entries in $\mathcal{S}$ would be zero. Explicitly storing zero-valued entries in a sparse tensor data structure is both undesirable and unnecessary; undesirable due to the high memory requirement of storing real-world tensors in general, and unnecessary because any non-explicitly–stored sparse tensor entries can simply be assumed to have the value zero. For these reasons, many real-world tensors are stored as sparse tensors [7].

2.2. Tensor Rank Decomposition

Tensor rank decomposition is similar to the matrix singular value decomposition, and can reveal latent information in data. Tensor rank decomposition is often referred to as canonical polyadic decomposition (CPD) or CANDECOMP/PARAFAC (CP). Computing CP for an N-mode tensor $\mathcal{X}$ yields an N-mode, rank-one tensor sum $\mathcal{M}$ approximating (or *modeling*) $\mathcal{X}$. The model $\mathcal{M}$ comprises R addends (with R as a parameter), where each addend is the outer product of N vectors (see Figure 1). Models are commonly represented using *Kruskal format* [8], written $\mathcal{X} \approx \mathcal{M} = [\![\lambda; A^{(1)}, A^{(2)}, \ldots, A^{(N)}]\!]$, where $A^{(k)} \in \mathbb{R}^{I_k \times R}$ (called a *factor matrix*) stores normalized versions of the R vectors associated with the k-th mode, and $\lambda \in \mathbb{R}^R$ (called the *weights* vector) stores the norms of the R vectors. This is because models represented in terms of their components often require less memory storage than explicitly storing the fully-evaluated tensor sum. (To see why, consider an $n \times n$ rank-one matrix which can be stored more efficiently as two vectors as opposed to storing a matrix.)

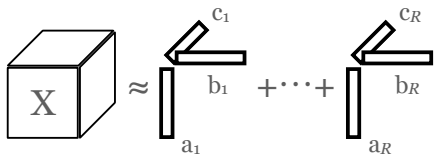

Figure 1: Three-dimensional ($N = 3$) tensor rank decomposition.

2.2.1. CP-APR. The primary algorithm for computing CP on a tensor containing sparse count data is CANDECOMP/PARAFAC via Alternating Poisson Regression (CP-APR), due to how a Poisson distribution suitably describes the random variation in such data [4]. There are three methods for computing CP-APR:

(i) Multiplicative Update (MU)
(ii) Projected Damped Newton for Row-based sub-problems (PDN-R)
(iii) Projected Quasi-Newton for Row-based sub-problems (PQN-R)

PDN-R and PQN-R are Newton-based methods which require fewer iterations to converge than MU because they use second-order information to solve independent row sub-problems, while MU uses a form of scaled steepest-descent with bound constraints over all rows during each iteration [9]. Each row sub-problem can be solved in parallel with PDN-R and PQN-R, but MU iterates towards a solution using dense matrix operations which can be implemented in parallel to achieve better overall performance than PDN-R and PQN-R on parallel systems. The main focus of our work is on parallel performance, so we focus on MU exclusively. (For a detailed discussion of the CP-APR MU algorithm, see [4].)

3. Sparse Tensor Storage Formats

This section presents an overview of the canonical and state-of-the-art sparse tensor storage formats, respectively coordinate (COO) (§3.1) and Adaptive Linearized Tensor Order (ALTO) (§3.2).

3.1. Coordinate (COO)

COO is the canonical sparse tensor storage format [8], and is similar to the sparse matrix storage format of the same name. Given an N-mode sparse tensor, COO lists each non-zero element with its corresponding N-dimensional coordinate, typically sorted by coordinates in lexicographical order. The two primary advantages to using the COO storage format are that it is intuitive, and that it straightforwardly allows iterating over non-zero entries and their corresponding coordinates. Performing a calculation on a sparse tensor stored in COO format involves iterating over each non-zero entry and using the corresponding coordinates for the entry to perform a desired operation. For example, consider a matrix-vector product using a 2-mode sparse tensor $\mathbf{X}$ and a dense (non-sparse) vector $\mathbf{y}$ and storing the result in dense vector $\mathbf{z}$: For each non-zero element v with coordinates (i, j) in $\mathbf{X}$, the mode-1 (row) coordinate i determines the participating row in $\mathbf{z}$, and the mode-2 (column) coordinate j determines the participating row in $\mathbf{y}$. Thus, the simplest approach iterates over each non-zero element v in $\mathbf{X}$, identifies its coordinates (i, j), then multiplies v by $\mathbf{y}_j$ and adds the result to $\mathbf{z}_i$.

The COO non-zero element list for an N-mode sparse tensor with V non-zero elements is easily represented in a computer program using one length-V value array and N length-V coordinate arrays, such that index i in each of the arrays identifies the value and corresponding coordinates for the i-th non-zero element. This representation uses $(N+1)V$ storage and allows straightforwardly iterating over the non-zero element list by iterating over the V array indices for all $N+1$ arrays simultaneously. Additionally, the representation allows parallel operation by dividing the V array indices between two or more processors.

The primary drawback to using COO is its explicit coordinate storage footprint. Consider a 3-mode sparse tensor $\mathcal{X}$ with dimensions $I_1 \times I_2 \times I_3$ and V non-zero elements. The storage for $\mathcal{X}$ using COO is $4V$, because we must store three coordinates and one value for each of the V non-zero elements. This means that the COO storage for $\mathcal{X}$ increases as V increases. V increases as the sparsity of $\mathcal{X}$ decreases, up to the bound of $I_1 I_2 I_3$ (i.e., a dense tensor). As V approaches $I_1 I_2 I_3$, the COO storage for $\mathcal{X}$ approaches $4\,(I_1 I_2 I_3)$, which exceeds the amount of storage required for storing $\mathcal{X}$ as a dense tensor (e.g. in multidimensional arrays) by a factor of 4. For COO to be at least as storage-efficient as multi-dimensional arrays, it must satisfy the condition $(N+1)V \leqslant \prod_{k=1}^{N} I_k \implies V \leqslant \prod_{k=1}^{N} I_k/(N+1)$, where the the sparse tensor has N dimensions and V non-zero entries. This means that COO storage efficiency requires the sparsity of a tensor not to exceed $1/(N+1)$.

A secondary drawback to COO is that iterating over the non-zero elements necessarily implies an ordering with respect to the non-zero element coordinates. State-of-the-art memory systems require regular memory accesses for maximizing cache utilization and achieving the best overall memory system performance. Thus, accessing a sparse tensor's non-zero entries in a different order than the order in which they are stored in memory leads to irregular memory accesses, which leads to poor cache utilization and poor overall memory system performance. Sparse tensor calculations like CP-APR MU require iterating over a sparse tensor's non-zero entries mode-wise, for all modes, which effectively results in irregular memory accesses for all modes except one of the modes.

As a simple example of the non-zero entry ordering issue, consider a 2-mode sparse tensor $\mathcal{X}$ (equivalently, a sparse matrix) with non-zero entries stored in coordinate lexicographical order or, equivalently, row-wise order. The first-mode coordinates are taken to be the row coordinates, so iterating over the non-zero entries is effectively row-wise iteration over the non-zero entries in $\mathcal{X}$. (This row-wise iteration is common in such basic linear algebra operations as the matrix-vector product.) If the second-mode coordinates are taken as column coordinates, then the current lexicographical ordering for the non-zero entries prohibits efficiently iterating over the non-zero entries column-wise. (An example of a column-wise iteration is the transposed-matrix times vector product.) Iterating over the non-zero entries column-wise in this scenario is inefficient because ordering the non-zero entries row-wise in memory generally means that the non-zero entries will not be ordered column-wise in memory, and iterating over the non-zero entries column-wise in this situation is what causes irregular memory accesses and reduced memory system performance.

Additionally, iterating over the non-zero entries column-wise in COO format is not even possible without sorting the non-zero entries by their column coordinates. In general, achieving a different ordering entails sorting the non-zero elements for each desired ordering. One approach for achieving this capability entails storing the non-zero elements in arbitrary order, then sorting the elements by mode, and storing the sorting results into N additional permutation arrays. This permutation array approach increases the COO storage requirement to $(2N + 1)V$ for a sparse tensor with N modes and V non-zero elements. For COO with permutation arrays to be at least as storage-efficient as using multi-dimensional arrays, the sparse tensor must satisfy the condition that the sparsity does not exceed $1/(2N + 1)$, where the sparse tensor has N dimensions. Thus, the COO format is an intuitive storage format with potential memory-performance and storage-footprint challenges for algorithm designers.

3.2. Adaptive Linearized Tensor Order (ALTO)

The state-of-the-art sparse tensor storage format is ALTO [10], which maps N-dimensional coordinates to linear indices such that non-zero elements near to each other

in a sparse tensor are near to each other in memory. Given an N-mode sparse tensor $\mathcal{X}$, ALTO essentially maps the coordinates for each non-zero element v in $\mathcal{X}$ to the set of natural numbers $\mathbb{N}$ by recursively dividing the multi-dimensional space occupied by the tensor into halves until locating v. Once v is located, the recursive division and subsequent choice of halves forms a bit sequence of choices which can be interpreted as a natural number. This natural number is the ALTO index for v.

Compared to COO, ALTO has lower memory usage due to storing a single linear index value for each non-zero entry, instead of N coordinate values. ALTO's lower memory usage leads to higher memory bandwidth utilization, because storing one index value per non-zero entry instead of N coordinate values reduces the required memory value accesses by $N-1$ per non-zero entry. This reduction in required memory accesses per non-zero entry frees crucial memory bandwidth and cache storage for other practical purposes, such as accessing additional non-zero entries or operand values during sparse tensor computations. The higher memory bandwidth utilization with ALTO is also due to the non-zero entries being near each other in memory when they are near each other in the N-dimensional tensor, thereby maintaining access locality. Another advantage compared to COO is that ALTO has dimension-agnostic storage requirements: ALTO stores V linear indices in a one-dimensional data structure and their V corresponding non-zero entries, meaning that ALTO requires $2V$ storage for an N-mode sparse tensor $\mathcal{X}$, irrespective of N.

ALTO Workload Partitioning and Scheduling. The ALTO approach differs from other sparse tensor storage formats which structure (*partition*) a given sparse tensor into coarse-grained storage units, because coarse-grained storage formats may not be amenable to balanced parallel workload *scheduling*. Coarse-grained sparse tensor storage formats partition a given sparse tensor into storage units containing more than one non-zero value, such as a blocks of non-zero values [11], or forests of height-N trees whose leaves are non-zero values [12]. These coarse-grained sparse tensor storage formats often partition a given sparse tensor into storage units of varying size due to the irregular sparsity patterns of real-world sparse tensors. A parallel sparse tensor calculation over storage units of varying size leads to workload imbalances of varying degree, which diminishes the overall effectiveness of parallel calculation because the storage unit assigned to one parallel process may be much larger than the storage unit assigned to another parallel process. This scenario over-burdens the former parallel process with work while starving the latter for work, hence the latter parallel process is not utilized to its full potential.

The ALTO sparse tensor storage format aims to expose fine-grained parallelism opportunities by partitioning a given sparse tensor in *storage units* at the granularity of individual non-zero values, each paired respectively with a one-dimensional coordinate encoding of their N-dimensional coordinates. This granularity level is similar to that of the COO sparse tensor format, which also structures a given sparse tensor in storage units of non-zero values, each paired with their respective N-dimensional coordinates. The advantage to structuring a given sparse tensor in storage units of individual pairs of non-zero values with their respective N-dimensional coordinate linear encodings (their respective *linear indices*) is that this allows sparse tensor calculations to be performed in parallel over storage units which are as fine-grained as possible. This enables a sparse tensor calculation to partition the set of all ALTO storage units into relatively balanced subsets which are amenable to relatively balanced parallel workloads.

4. CP-APR MU Algorithm

This section starts with an algorithmic overview of CP-APR MU (§4.1), continues into sparse tensor implementation details (§4.2), and finishes with a discussion on performance considerations and optimization opportunities (§4.3). For the mathematical derivation of the CP-APR MU algorithm, see Chi and Kolda [4].

4.1. CP-APR MU Algorithm Overview

A program listing for the general CP-APR MU algorithm is shown in Algorithm 1, using the notation shown in Table 1. The CP-APR MU algorithm takes an N-mode input tensor $\mathcal{X}$ and iterates towards an approximate model tensor $\mathcal{M}$ in two nested stages, called *outer* and *inner*, where the *outer* iteration updates each of the factor matrices in round-robin fashion, and the *inner* iteration calculates successive updates to the current factor matrix. In each iteration of the *outer* stage, CP-APR MU updates the N model factor matrices, $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}$, as follows: If at least one *outer* iteration has completed, then the algorithm searches through the current model factor matrix $\mathbf{A}^{(n)}$, $1 \leqslant n \leqslant N$, for values which are smaller than a minimum value parameter $\kappa_{\min}$. This is because values near zero may interfere with solution convergence. These values are called *inadmissible zeros*, and if an inadmissible zero is detected, it is shifted by an adjustment value parameter, κ_{adj}. After adjusting for inadmissible zeros, $\mathbf{A}^{(n)}$ is scaled by the weight vector λ, resulting in the current working model factor matrix $\mathbf{B}$. The last step before entering the *inner* iterative stage is to calculate the intermediate $\Pi^{\top}$ matrix as a chained column-wise Kronecker product over all model factor matrices excluding $\mathbf{A}^{(n)}$.

In each iteration of the *inner* iterative stage, the current model factor matrix $\mathbf{A}^{(n)}$, $1 \leqslant n \leqslant N$, undergoes a series of multiplicative updates. Each multiplicative update in the series requires calculating the intermediate $\mathbf{\Phi}$ matrix as an element-wise division between the mode-n flattening of $\mathcal{X}$ (i.e., $\mathbf{X}_{(n)}$) and the matrix-matrix product of $\mathbf{B}$ and $\mathbf{\Pi}$, all followed by a matrix multiplication by $\mathbf{\Pi}^{\top}$. Then, $\mathbf{B}$ is compared element-wise with $\mathbf{\Phi}$ against the convergence tolerance τ: If the convergence tolerance is satisfied, then the *inner* stage ceases immediately. Otherwise, $\mathbf{B}$ is multiplied element-wise by $\mathbf{\Phi}$ (i.e., the *multiplicative update*) and the *inner* iterative stage repeats. After satisfying the

convergence tolerance, or otherwise completing the maximum desired iterations in the *inner* iterative stage, the *outer* iterative stage returns updated weights to λ from $\mathbf{B}$ and normalizes $\mathbf{A}^{(n)}$ using the updated λ. The algorithm concludes when all N tensor modes have converged simultaneously, or otherwise after completing the maximum desired iterations in the *outer* iterative stage.

Algorithm 1 CP-APR MU calculation.

Given an N-mode tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and an initial guess for a model $\mathcal{M} = \left\{\lambda; \mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}\right\}$:

for (maximum *outer* iterations) **do**
  $\texttt{is\_converged} \leftarrow \text{true}$
  for $n \leftarrow 1, 2, \ldots, N$ **do**
    $\mathbf{S} \leftarrow \mathbf{0}$
    if (completed at least one *outer* iteration) **then**
      if $\mathbf{A}^{(n)}_{i,r} < \kappa_{min}$ and $\mathbf{\Phi}^{(n)}_{i,r} > 1$ **then**
        $\mathbf{S}_{i,r} \leftarrow \kappa_{\text{adj}}$
      end if
    end if
    $\mathbf{B} \leftarrow \left(\mathbf{A}^{(n)} + \mathbf{S}\right)\mathbf{\Lambda}$
    $\mathbf{\Pi}^{\top} \leftarrow \circledast_{k \neq n} \mathbf{A}^{(k)}$
    for (maximum *inner* iterations) **do**
      $\mathbf{\Phi} \leftarrow \left(\mathbf{X}_{(n)} \oslash \max^{\circ}\left(\mathbf{B}\mathbf{\Pi}, \epsilon\right)\right)\mathbf{\Pi}^{\top}$
      if $\max_{i,r}\left|\min\left(\mathbf{B}_{i,r}, 1 - \mathbf{\Phi}^{(n)}_{i,r}\right)\right| < \tau$ **then**
        break
      end if
      $\texttt{is\_converged} \leftarrow \text{false}$
      $\mathbf{B} \leftarrow \mathbf{B} \circ \mathbf{\Phi}$
    end for
    $\lambda \leftarrow \mathbf{1}^{\top}\mathbf{B}$
    $\mathbf{A}^{(n)} \leftarrow \mathbf{B}\mathbf{\Lambda}^{-1}$
  end for
  if $\texttt{is\_converged} = \text{true}$ **then**
    break
  end if
end for

4.2. CP-APR MU Sparse Tensor Implementation

Most real-world tensors contain a massive amount of multidimensional data, which would have a storage footprint on the order of exabytes (e.g., 3.2×10^{20} bytes for *LBNL-Network* [7]) if explicitly stored as a dense multidimensional array. This storage requirement is infeasible on most computing systems. In addition, these data are sparse, meaning that the majority of the points in the multidimensional space have the value zero. Thus it is practical to store only the non-zero values of a tensor (e.g., 13.5×10^6 bytes for *LBNL-Network*) with their corresponding coordinates.

State-of-the-art implementations of sparse CP-APR MU iterate over the non-zero values of a given input sparse tensor. This is because sparse tensor storage formats generally do not prescribe a mapping from coordinates to values, as is the case with the sparse tensor storage formats described previously (COO and ALTO). Instead, each of these sparse tensor storage formats stores metadata with the non-zero values for the purpose of retrieving the coordinates corresponding with each non-zero value: For COO, the metadata are simply the coordinate N-tuples corresponding with each non-zero value. For ALTO, the metadata are linear indices representing encoded versions of the coordinate N-tuples corresponding with each non-zero value.

Each format stores metadata for calculating the coordinate of a non-zero value along with the non-zero value itself. This means that the coordinate for a given non-zero value cannot be retrieved without first locating the non-zero value in the sparse tensor storage format. The calculations for CP-APR MU depend on accessing all of the coordinates corresponding with each of the non-zero values. Thus, the most direct way to access all of the coordinates stored in one of the sparse tensor storage formats described previously is to iterate over the stored non-zero values and retrieve the coordinate corresponding with each non-zero value. The CP-APR MU algorithm then accesses participating rows and columns in the working model factor matrix $\mathbf{B}$ and the intermediate calculation matrices $\mathbf{\Pi}$ and $\mathbf{\Phi}$, because these are stored densely (e.g. as typical two-dimensional arrays). Note that only the rows of $\mathbf{\Pi}$ which correspond with coordinates for non-zero values in the sparse tensor are required for calculating CP-APR MU, so typically only those rows are calculated.

Program listings for sparse CP-APR MU $\mathbf{\Pi}$ and $\mathbf{\Phi}$ calculations are shown in Algorithms 2 and 3, respectively.

Algorithm 2 Sparse CP-APR MU mode-n $\mathbf{\Pi}$ calculation.

for $k \leftarrow 1, 2, \ldots, N$ **do**
  if $k \neq n$ **then**
    for $i \leftarrow 1, 2, \ldots, \text{nnz}$ **do**
      $\texttt{row} \leftarrow \texttt{coord}_{n,\mathcal{X},i}$
      for $r \leftarrow 1, 2, \ldots, R$ **do**
        $\mathbf{\Pi}_{i,r} \leftarrow \mathbf{\Pi}_{i,r} * \mathbf{A}^{(k)}_{\texttt{row},r}$
      end for
    end for
  end if
end for

Algorithm 3 Sparse CP-APR MU mode-n $\mathbf{\Phi}$ calculation.

for $i \leftarrow 1, 2, \ldots, \text{nnz}$ **do**
  $\texttt{row} \leftarrow \texttt{coord}_{n,\mathcal{X},i}$
  $\texttt{temp} \leftarrow 0$
  for $r \leftarrow 1, 2, \ldots, R$ **do**
    $\texttt{temp} \leftarrow \texttt{temp} + \mathbf{B}_{\texttt{row},r} * \mathbf{\Pi}_{i,r}$
  end for
  $\texttt{temp} \leftarrow \mathcal{X}_i / \max(\texttt{temp}, \epsilon)$
  for $r \leftarrow 1, 2, \ldots, R$ **do**
    $\mathbf{\Phi}_{\texttt{row},r} \leftarrow \mathbf{\Phi}_{\texttt{row},r} + \texttt{temp} * \mathbf{\Pi}_{i,r}$
  end for
end for

4.3. Performance Analysis and Optimization

The CP-APR MU Φ calculation is shown in Equation 4.3:

$$\mathbf{\Phi}^{(n)} \leftarrow \left(\mathbf{X}_{(n)} \oslash \left(\max\left(\mathbf{B}\mathbf{\Pi}, \epsilon\right)\right)\right)\mathbf{\Pi}^{\top} \tag{1}$$

The $\mathbf{X}_{(n)}$ matrix is a flattened representation of the sparse tensor $\mathcal{X}$, but $\mathbf{X}_{(n)}$ need not be stored explicitly because we can access all of its elements by accessing them in the sparse tensor $\mathcal{X}$. The $\mathbf{\Pi}^{\top}$ matrix is a chained Khatri-Rao product of size $\prod_{k\neq n} I_k \times R$, which if stored explicitly would require as much storage as a densely-stored tensor $\mathcal{X}$. If $\mathcal{X}$ is too large to store densely, then $\mathbf{\Pi}^{\top}$ is also too large to store explicitly. As a concrete example, consider a four-mode tensor $\mathcal{S}$ of size $1000 \times 1000 \times 1000 \times 1000$. The $\mathbf{\Pi}^{\top}$ matrix for $\mathcal{S}$ is size $10^9 \times R$, which requires $10^9 \times 10 \times 4$ bytes $= 40\,\text{GiB}$ storage when using $R = 10$ and storing single-precision floating-point values. Conveniently, computing MU on a sparse tensor $\mathcal{X}$ does not require the entire $\mathbf{\Pi}^{\top}$ matrix. This is because computing MU on a sparse tensor $\mathcal{X}$ requires only the rows of $\mathbf{\Pi}^{\top}$ which correspond with nonzero entries in $\mathcal{X}$ [4], reducing the storage requirement greatly: Suppose $\mathcal{S}$ has 1 M non-zero elements. The necessary rows of $\mathbf{\Pi}$ require $1 \times 10^6 \times 10 \times 4$ bytes $= 40\,\text{MiB}$ storage (single-precision), which is a 1000-fold reduction in memory storage requirements when compared with forming the entire $\mathbf{\Pi}^{\top}$ matrix. State-of-the-art MU implementations calculate only the necessary rows of $\mathbf{\Pi}^{\top}$ [2], which allows computing MU on larger inputs than would be possible otherwise (due to memory space constraints).

The sparse tensor storage formats typically store non-zero elements with their corresponding coordinates as a list. For this reason, sparse tensor decomposition algorithms typically iterate over the list of non-zero elements, rather than iterating in a coordinate-centric fashion. Calculating $\mathbf{\Phi}^{(n)}$ in parallel can manifest race conditions, particularly when using sparse tensor storage. This is because two non-zero elements which share the same coordinate for mode n will correspond with updating the same row in $\mathbf{\Phi}^{(n)}$, so these updates must be serialized to maintain correctness. A well-known strategy for serializing updates to shared data is by using atomic operations, which help to ensure exclusive access to the shared data. This atomic operation strategy helps in maintaining program correctness, but diminishes the benefits of parallel execution due to serialization. For example, if two threads are assigned many non-zeros which update the same row in $\mathbf{\Phi}^{(n)}$, then processing those non-zeros is effectively sequential. A variation on this approach entails sorting the non-zero elements by mode n (ignoring the remaining coordinates) so that those which update the same row in $\mathbf{\Phi}^{(n)}$ are stored contiguously. This means that a thread can skip atomic operations for the portions of its assigned non-zero elements which are known not to share a row coordinate in $\mathbf{\Phi}^{(n)}$ with any other thread. For example, if a thread is assigned non-zero elements which update row $r-1, r, r+1$ in $\mathbf{\Phi}^{(n)}$, then all non-zeros in $\mathcal{X}$ which update row r are solely processed by this thread, meaning that this thread can avoid atomic operations for row r. Sorting will be required for each mode n in order to use this atomic-avoiding approach, because $\mathbf{\Phi}^{(n)}$ is calculated for each mode. High performance MU implementations perform the sorting in advance, storing permuted indices in separate arrays for each mode, or use other approaches to reduce parallel contention [2], [10].

5. Methods

We first describe the methods we use in evaluating parallel performance. As a first step, we chose STREAM for two reasons: (i) the MTTKRP and MU are bandwidth-bound, as are the STREAM operations, and (ii) STREAM operations can be used as building blocks for the MTTKRP and for MU.

5.1. CP-ALS and MTTKRP

We will limit the following discussion to three-dimensional (three-way) tensors, although the discussion generalizes to arbitrary dimension. Given a three-way tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$, the CP-ALS algorithm computes a rank-R model tensor $\mathcal{M}$, consisting of factor matrices $A \in \mathbb{R}^{I_1 \times R}$, $B \in \mathbb{R}^{I_2 \times R}$, and $C \in \mathbb{R}^{I_3 \times R}$, that approximates $\mathcal{X}$. Using typical tensor notation, $\mathcal{X} \approx \mathcal{M} = [\![A, B, C]\!]$. In CP-ALS, MTTKRP is often the performance bottleneck, consuming over 90% of the total compute time [1].

MTTKRP consists of a few simple operations. For a sparse tensor stored in COO format and factors stored as dense matrices, given a non-zero element in $\mathcal{X}$ with indices (i, j, k) and value v, the following operations are required (with temporary variable T):

$$T(:) \leftarrow B(j,:) * C(k,:) \quad \text{element-wise product} \tag{2}$$

$$T(:) \leftarrow v * T(:) \quad \text{scale} \tag{3}$$

$$A(i,:) \leftarrow A(i,:) + T(:) \quad \text{element-wise add} \tag{4}$$

where $A(i,:)$, $B(j,:)$ and $C(k,:)$ correspond to the rows of the factor matrices. This is repeated for every non-zero element in $\mathcal{X}$. A program listing for sparse tensor mode-1 MTTKRP is shown in Algorithm 4.

Algorithm 4 Sparse tensor mode-1 MTTKRP calculation.

Input: A three-dimensional sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, and matrices $\mathbf{B} \in \mathbb{R}^{I_2 \times R}$ and $\mathbf{C} \in \mathbb{R}^{I_3 \times R}$.
Output: A matrix $\mathbf{A} \in R^{I_1 \times R}$.

1: **for** $i \leftarrow 1, 2, \ldots, \text{nnz}$ **do**
2: $\quad i_1 \leftarrow \texttt{coord}_{1,\mathcal{X},i}$
3: $\quad i_2 \leftarrow \texttt{coord}_{2,\mathcal{X},i}$
4: $\quad i_3 \leftarrow \texttt{coord}_{3,\mathcal{X},i}$
5: $\quad$ **for** $r \leftarrow 1, 2, \ldots, R$ **do**
6: $\quad\quad \mathbf{A}_{i_1,r} \leftarrow \mathbf{A}_{i_1,r} + \mathcal{X}_i \mathbf{B}_{i_2,r} \mathbf{C}_{i_3,r}$
7: $\quad$ **end for**
8: **end for**

5.2. CP-APR MU and Phi (Φ)

The CP-APR algorithm is similar to CP-ALS, except that it is better suited for modeling count-based data. In the MU variant of CP-APR, calculating a matrix $\mathbf{\Phi}^{(n)}$ for each tensor mode n is often the performance bottleneck, as shown in Figure 2.

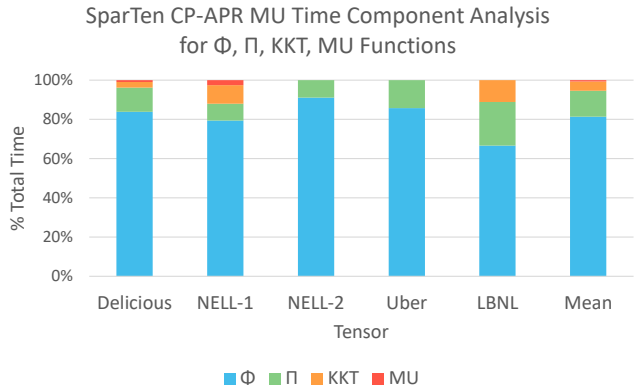

Figure 2: Execution time analysis for SparTen CP-APR MU kernels on five tensors from FROSTT [7]. The four kernels are for computing Φ, Π, KKT conditions, and multiplicative update (MU).

Computing the Φ matrix consists of a few simple operations. For a three-way sparse tensor stored in COO format and factors stored as dense matrices, given a non-zero element in $\mathcal{X}$ with indices (i, j, k) and value v, the following operations are required (with temporary variable t):

$$t \leftarrow \text{sum}\left(A(i,:) \circ B(j,:) \circ C(k,:)\right) \quad (5)$$

three-way dot product

$$\Phi(i,:) \leftarrow \Phi(i,:) + v/t * \left(B(j,:) \circ C(k,:)\right) \quad (6)$$

scaled element-wise product sum

where $\Phi(i,:)$, $A(i,:)$, $B(j,:)$, and $C(k,:)$ correspond to the rows of the matrices. This is repeated for every non-zero element in $\mathcal{X}$.

5.3. Challenges

We can see from the above equations that MTTKRP and MU exhibit *low arithmetic intensity* (i.e., they are memory bandwidth-bound). Additionally, the last steps (Equations 4 and 6) introduce a *race condition* when multi-threaded, making the kernel sensitive to how work is distributed among threads on a parallel system. For example, if two threads work on non-zero elements with the same i index, the updates to $A(i,:)$ and $\Phi(i,:)$ need to be serialized.

However, we can also see that these operations are similar to those found in the STREAM benchmark, which also features element-wise product and element-wise add. Therefore, we use the STREAM benchmark as a proxy for the MTTKRP and MU kernels, and the MTTKRP and MU kernels as proxies for the full CP-ALS and CP-APR algorithms, respectively. We based our MTTKRP on the Parallel Sparse Tensor Algorithm Benchmark Suite (PASTA) MTTKRP [11], and our MU on SparTen [2].

5.4. Implementation

Implementing Kokkos parallel constructs within an existing code base is a straightforward process of refactoring only targeted code regions to utilize the parallel code execution and data management in the Kokkos programming model. We first identify parallel regions in the code, such as those within existing OpenMP `#pragma` statements, and replace them with Kokkos `parallel_for` dispatch while incorporating the loop body into a C++ lambda expression. The next step is to refactor nested parallel regions and to store data in abstractions called *Views*, after which the code is completely portable to any back-end supported by the Kokkos library. Nested parallel regions map to SIMD instructions when compiling with Kokkos for CPU and to thread blocks for GPU targets. Note that OpenMP 4.5+ supports offloading to GPU devices [13], but we use Kokkos for performance portability due to its ability to handle data layout efficiently for both dense and sparse operations.

In the case of MU, we chose to compare SparTen [2] with ALTO [10], whose key differences are as follows: SparTen is implemented using Kokkos and using the COO sparse tensor storage format, and it computes the Π matrix explicitly in each outer iteration of CP-APR MU. ALTO is implemented using OpenMP and using the ALTO storage format, and it computes Π values *on-the-fly* as needed for the Φ computation. As mentioned previously, the Π matrix is often quite large and its values must be recomputed in each outer iteration of the CP-APR MU algorithm. Computing Π values on-the-fly reduces the overall memory footprint due to not explicitly storing the Π matrix, so we opted to implement on-the-fly calculation in SparTen. Similarly, we implemented the COO storage format for CP-APR MU in ALTO. Note that ALTO offers a choice of two kernel implementations for computing Φ: An atomic operation–based kernel (like SparTen) which prevents race conditions when updating Φ in parallel, and a pull-based accumulation kernel which uses a two-stage approach for avoiding conflicting parallel updates entirely. We implemented the COO storage format for both kernels, herein referred to as Atomic Edition (AE) and Pull-based Edition (PE), respectively. We refer to the modified versions of ALTO and SparTen respectively as *COO-format ALTO* (COO ALTO) and *reduced-footprint SparTen* (RF SparTen). Our intent with these modified versions is to isolate Kokkos as the key distinguishing component by providing a fair comparison between the MU kernels and storage formats in SparTen and ALTO.

6. Experimental Results

Now we provide details on our test kernels (§6.1) and test systems and data (§6.2), and our analysis (§6.3).

6.1. Test Kernels

For measuring baseline system memory performance we employ a tuned version of the STREAM benchmark, which aims to measure main memory performance by exhausting the cache hierarchy with simple operations on large arrays. These include simple copy, scale, add, and triad operations. For a more representative tensor decomposition-related kernel, we use our enhanced PASTA sparse matrixized tensor times Khatri-Rao product (MTTKRP) benchmark. Finally, for a fully representative tensor decomposition kernel, we use our reduced-footprint CP-APR MU $\mathbf{\Phi}$ calculation.

We demonstrate the performance portability of our Kokkos-enhanced STREAM and MTTKRP benchmarks, and our reduced-footprint CP-APR MU, by comparing their performance against (i) hand-tuned benchmarks written in their native languages (e.g., CUDA), and (ii) peak system memory bandwidth (for STREAM and MTTKRP) on a range of different systems using both synthetic and real-world data, including 3-D and 4-D tensors.

6.2. Test Systems and Data

We evaluate our kernels on the nine systems shown in Table 2, which includes five CPU systems and four GPU systems. For the kernels in the STREAM benchmark, we use up to $500M$ elements per array. For the MTTKRP and MU kernels, we use the real-world sparse tensors from the FROSTT [7] website shown in Table 3.

TABLE 2: Test Systems

Type	Name	# Cores
CPU	IBM POWER9	20
CPU	Intel Xeon Gold 6140	2 × 18
CPU	AMD EPYC 7401	2 × 24
CPU	AMD EPYC 7452	2 × 32
CPU	Fujitsu A64FX	48
GPU	AMD Vega MI25	4096
GPU	AMD Vega MI50	3840
GPU	Nvidia V100	5120
GPU	Nvidia A100	6912

TABLE 3: Test Data

Tensor	Dimensions	NNZ
Chicago-crime	6.2K × 24 × 77 × 32	5.3M
NELL-2	12.1K × 9.2K × 28.8K	76.9M
NIPS	2.5K × 2.9K × 14.0K × 17	3.1M
Uber	183 × 24 × 1.1K × 1.7K	3.3M

6.3. Analysis

Figure 3 shows the achieved bandwidth from various STREAM operations, and Figure 4 shows the speedup over hand-tuned benchmarks (i.e., STREAM for CPUs and GPU-STREAM for GPUs). We achieve performance comparable to hand-tuned code (0.64×–1.66× speedup) for all STREAM operations, demonstrating that for simple kernels, Kokkos offers a good portability on different architectures.

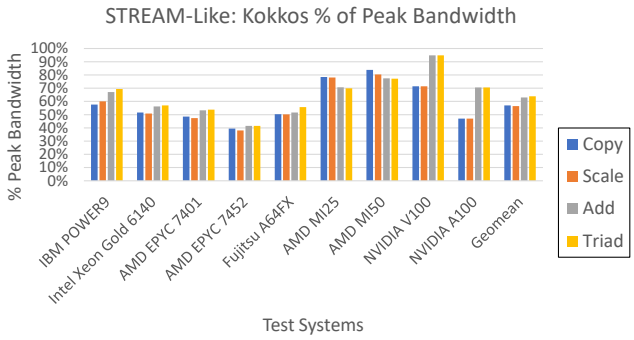

Figure 3: Kokkos-enhanced STREAM, percentage of system peak obtained.

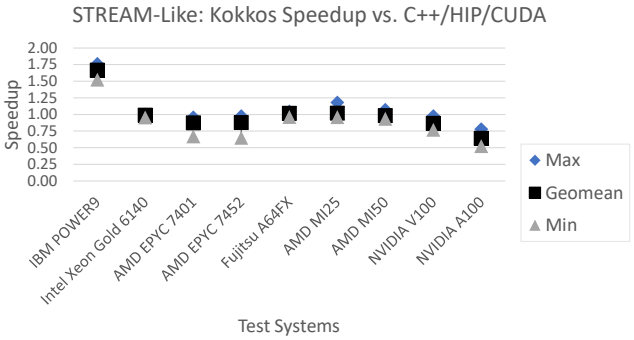

Figure 4: Kokkos-enhanced STREAM speedup over hand-tuned code (logarithmic scale).

Similarly, Figures 5–6 show the achieved bandwidth and speedup for the MTTKRP benchmark. We achieve *superior* performance on CPUs, and lower (0.76×–0.91× speedup) but comparable performance on Nvidia GPUs. Speedup numbers on AMD GPUs are missing due to PASTA supporting only Nvidia GPUs, which further illustrates the advantage of using Kokkos—there is no need to implement yet another kernel for a different system. The lower performance for AMD GPUs likely comes from the lack of hardware atomic operation units for double-precision data.

Figures 7, 8, and 9 show runtimes for the CP-APR MU Phi (Φ) kernel for our reduced-footprint SparTen (RF SparTen), and for both of our COO-format ALTO (COO ALTO) implementations, respectively. Figures 10–11 show the speedup of COO ALTO AE (using atomic operations) vs. RF SparTen, and COO ALTO PE (using pull-based accumulation) vs. RF SparTen, respectively. Our RF SparTen achieves lower to comparable performance to our COO ALTO AE on CPUs (0.47×–0.96× speedup), but RF SparTen performance falls behind that of COO ALTO PE on CPUs, due to the latter avoiding expensive atomic operations. Speedup numbers are missing for the IBM and

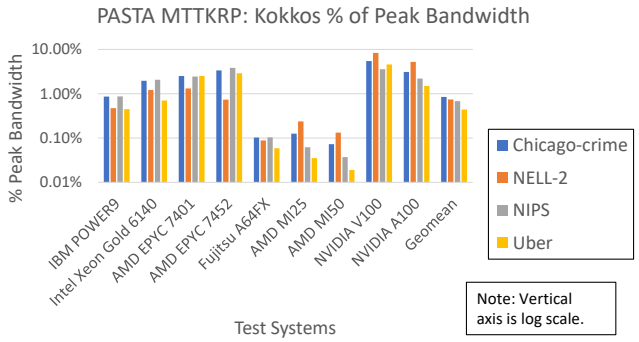

Figure 5: Kokkos-enhanced PASTA MTTKRP, percentage of system peak obtained (logarithmic scale).

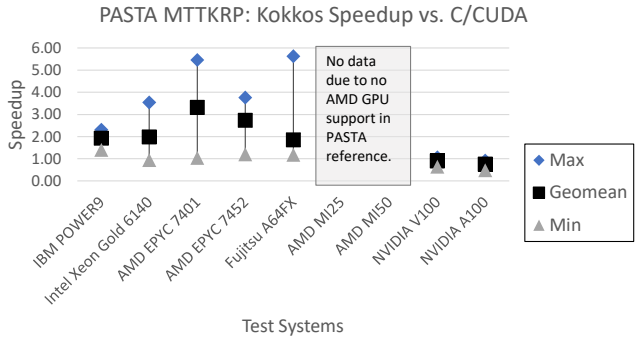

Figure 6: Kokkos-enhanced PASTA MTTKRP speedup over hand-tuned code.

Fujitsu CPUs due to issues regarding lacking support at the instruction level (bit-wise scatter-gather) for ALTO's coordinate linearization and delinearization. While our augmented COO ALTO does not use the (linearized) ALTO format during computation, it uses the ALTO format generation process for partitioning and scheduling the nonzeros stored in COO format. Speedup numbers are missing for GPUs due to ALTO supporting only CPUs, and although SparTen (via Kokkos) has the advantage of supporting GPUs, due to time constraints we did not conduct GPU experiments with SparTen for this work.

Our results show that Kokkos demonstrates good performance portability for essential sparse tensor decomposition kernels across the range of CPU and GPU systems tested when compared with *algorithmically similar* platform-specific hand-tuned code. That is, the Kokkos kernel implementations perform comparably to the platform-specific kernel implementations *except* in the case where the platform-specific kernel also uses a substantially different algorithmic design.

7. Conclusion

Our efforts in this study demonstrate the feasibility of writing performance portable tensor decomposition algorithms using the Kokkos Core library that can achieve hand-tuned performance on a range of systems using a single implementation. We achieve comparable performance on CPUs and GPUs for simple array operations and superior performance on CPUs for the MTTKRP kernel. However, additional tuning is required on GPUs for the more complicated MTTKRP kernel due to the large number of threads required to saturate performance and the use of

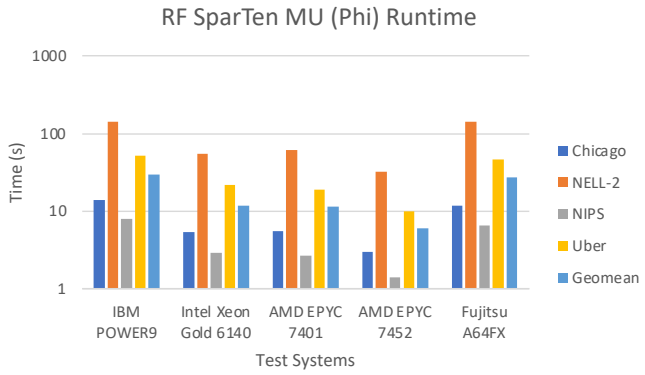

Figure 7: Reduced-Footprint (RF) SparTen CP-APR MU Phi (Φ) kernel runtime (logarithmic scale).

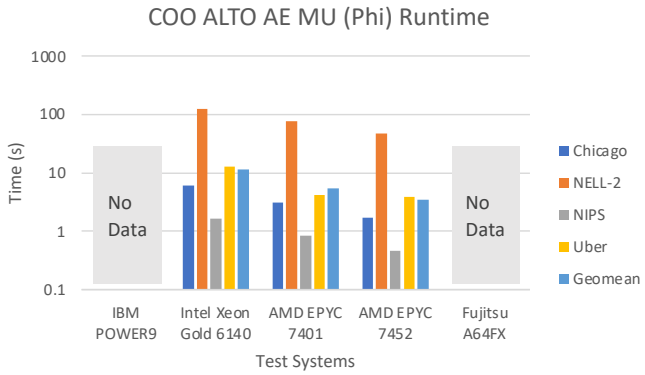

Figure 8: COO-format ALTO CP-APR MU Phi (Φ) kernel, Atomic Edition (AE) runtime (logarithmic scale).

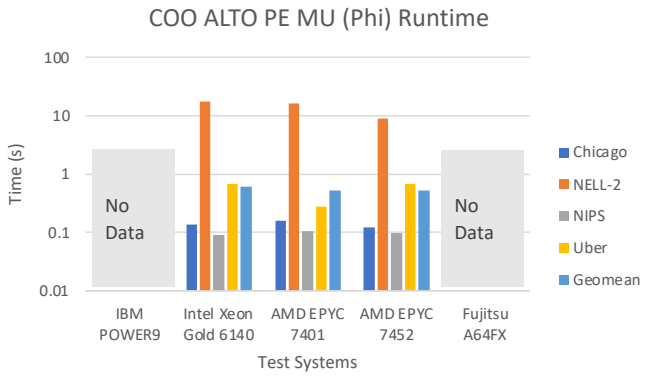

Figure 9: COO-format ALTO CP-APR MU Phi (Φ) kernel, Pull-based Edition (PE) runtime (logarithmic scale).

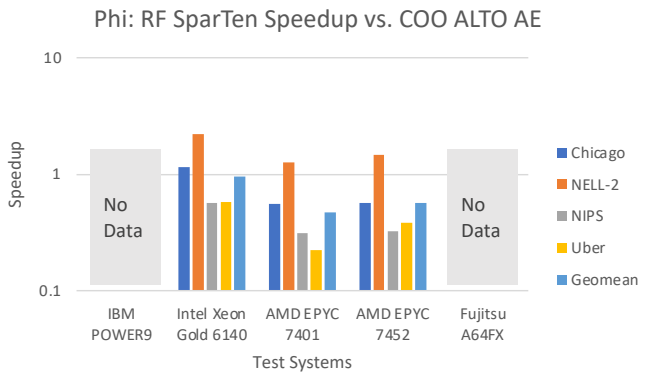

Figure 10: COO-format ALTO CP-APR MU Phi (Φ) kernel, Atomic Edition (AE) speedup vs. reduced-footprint SparTen CP-APR MU Phi (Φ) kernel (logarithmic scale).

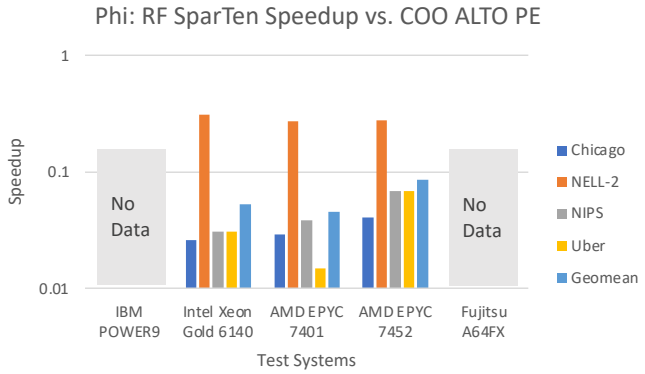

Figure 11: COO-format ALTO CP-APR MU Phi (Φ) kernel, Pull-based Edition (PE) speedup vs. reduced-footprint SparTen CP-APR MU Phi (Φ) kernel (logarithmic scale).

expensive atomic operations. We see that Kokkos maintains comparable performance portability in the case of the application-level CP-APR MU Phi (Φ) kernel, provided that the comparison is between two similar algorithms. While the Kokkos implementation of the CP-APR MU Phi (Φ) kernel fell behind that of the ALTO pull-based accumulation kernel, this was likely due to the Kokkos implementation using computationally expensive atomic operations while the ALTO pull-based implementation did not. Our claim is that the choice of algorithm has a larger effect on performance than does the choice of using Kokkos vs. platform-specific hand-tuned code, and that a Kokkos implementation has the advantage of being portable.

One possible direction of future work is comparing performance for the CP-APR MU kernel on IBM and Fujitsu CPUs (and on GPU systems) between a Kokkos implementation and a platform-specific implementation. Another possible direction is investigating whether a CP-APR MU pull-based accumulation kernel implemented using Kokkos would be performance portable.

Acknowledgment

Portions of this work were funded during an internship at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., which is a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. These funded portions were published in the IEEE High Performance Extreme Computing 2021 Proceedings, and copies are accessible under identifier SAND2021-10690 C on OSTI.gov.

References

[1] E. T. Phipps and T. G. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019. [Online]. Available: https://doi.org/10.1137/18M1210691

[2] K. Teranishi, D. M. Dunlavy, J. M. Myers, and R. F. Barrett, "Sparten: Leveraging kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to poisson data," *IEEE High Performance Extreme Computing Conference*, vol. 0, no. 0, p. 0, 2020.

[3] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Proc. Extreme Scaling Workshop*, 2013, pp. 18–24.

[4] E. C. Chi and T. G. Kolda, "On tensors, sparsity, and nonnegative factorizations," *SIAM Journal on Matrix Analysis and Applications*, vol. 33, no. 4, pp. 1272–1299, 2012.

[5] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *Proc. ISC High Performance*, 2016, pp. 489–507.

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

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

[8] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2008. [Online]. Available: https://doi.org/10.1137/060676489

[9] S. Hansen, T. Plantenga, and T. G. Kolda, "Newton-based optimization for Kullback-Leibler nonnegative tensor factorizations," *Optimization Methods and Software*, vol. 30, no. 5, pp. 1002–1029, April 2015.

[10] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: Adaptive linearized storage of sparse tensors," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 404–416. [Online]. Available: https://doi.org/10.1145/3447818.3461703

[11] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker, "Pasta: A parallel sparse tensor algorithm benchmark suite," arXiv:1902.03317, 2019.

[12] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 61–70. [Online]. Available: https://doi.org/10.1109/IPDPS.2015.27

[13] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Evaluating support for openmp offload features," in *International Conference on Parallel Processing Companion*, 2018.