SparTen: Leveraging Kokkos for On-node Parallelism in a Second-Order Method for Fitting Canonical Polyadic Tensor Models to Poisson Data

Keita Teranishi
Sandia National Laboratories
knteran@sandia.gov

Daniel M. Dunlavy
Sandia National Laboratories
dmdunla@sandia.gov

Jeremy M. Myers
College of William and Mary
jermyer@sandia.gov

Richard F. Barrett
Sandia National Laboratories
rfbarre@sandia.gov

***Abstract*—Canonical Polyadic tensor decomposition using alternate Poisson regression (CP-APR) is an effective analysis tool for large sparse count datasets. One of the variants using projected damped Newton optimization for row subproblems (PDNR) offers quadratic convergence and is amenable to parallelization. Despite its potential effectiveness, PDNR performance on modern high performance computing (HPC) systems is not well understood. To remedy this, we have developed a parallel implementation of PDNR using Kokkos, a performance portable parallel programming framework supporting efficient runtime of a single code base on multiple HPC systems. We demonstrate that the performance of parallel PDNR can be poor if load imbalance associated with the irregular distribution of nonzero entries in the tensor data is not addressed. Preliminary results using tensors from the FROSTT data set indicate that using multiple kernels to address this imbalance when solving the PDNR row subproblems in parallel can improve performance, with up to 80% speedup on CPUs and 10-fold speedup on NVIDIA GPUs.**

***Index Terms*—tensor decomposition, Poisson factorization, Kokkos, multicore, GPU**

I. Introduction

Tensor decomposition based on the canonical polyadic (CP) model is a useful tool for analyzing multiway data sets, often providing insight that cannot be described using the raw data alone [1]. For sparse count data, alternating Poisson regression (CP-APR) [2], [3] is an effective approach, solving a bound-constrained nonlinear optimization problem to compute the decomposition. Hansen *et al.* have demonstrated that computing CP-APR tensor decompositions using the method called Projected Damped Newton for Row subproblems (PDNR) provides quadratic convergence and could leverage parallel computation to solve independent subsets of the decomposition [3]. However, implementation and performance analysis of the PDNR method on high performance computing (HPC) systems has received little attention compared to the fixed-point iteration method based on Multiplicative Updates [2], which can be implemented with data-parallel linear algebra operators [4]–[6].

In this paper, we examine the implementation of the PDNR method for HPC systems with a focus on parallelization and portability for a variety of node architectures. Our study is performed with SparTen[1], a C++ sparse, node-parallel CP-APR solver package implemented using the performance-portable hardware abstraction library called Kokkos [7]. SparTen supports three different variants of CP-APR tensor decomposition algorithms [2], [3] and provides efficient computation on major HPC node architectures including x86-multicore, ARM, and NVIDIA GPUs. In this paper, we focus on the PDNR implementation in SparTen, which uses a damped Newton optimization solver kernel written with Kokkos' parallelization primitives, allowing us to maintain a single code base that performs well independent of the underlying architecture. Despite the usefulness of Kokkos, our initial PDNR implementation, based on concurrent execution of many single-kernel Newton optimization solver instances, exhibits load imbalance with real-world tensor data due to skewed distributions of nonzero entries across the rows of the unfolded sparse tensors. We mitigate this imbalance with a hybrid multi-kernel approach that manages parallel computation based on the workload in solving individual row subproblems. Our preliminary results indicate that our multi-kernel approach achieves up to 80% speedup on CPUs and ten-fold speedups on GPUs over the initial single-kernel approach.

The rest of the paper is organized as follows. A summary of sparse CP-APR tensor decomposition, including related work, and the PDNR algorithm is presented in Section II. The design and implementation of PDNR using Kokkos, along with its performance considerations, are discussed in Section III. Performance results are presented and discussed in Section IV. We present a summary of findings and possible future work in Section V.

II. Background and Related Work

In this section, we review sparse CP-APR PDNR algorithm, Kokkos, and related work. For a comprehensive review of tensors, see [1]. The algorithms for CP-APR tensor decomposition were originally developed in [2], [3]. The notation in this paper follows those used in [3].

A. Canonical Polyadic Alternating Poisson Regression

A tensor is a multi-dimensional array or multi-dimensional mapping of data. An N-way tensor $\mathcal{X}$ has size $I_1 \times I_2 \times \cdots \times I_N$. Figure 1 illustrates the canonical polyadic (CP) decomposition of a 3-way data tensor, where the model is expressed as

[1]Available at `http://gitlab.com/tensors/sparten`

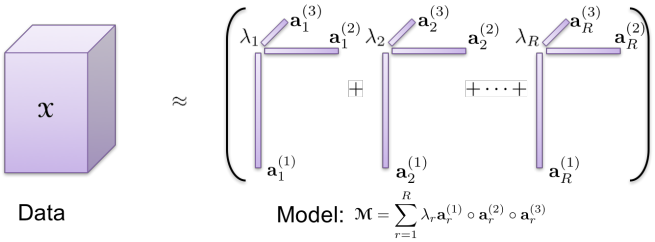

Fig. 1. CP decomposition of a 3-way tensor into R components.

a weighted sum of tensors formed using outer products of vectors, or *components*. A rank-R CP decomposition [8], [9] of an N-way tensor $\mathcal{X}$ is defined as:

$$\mathcal{X} \approx \mathcal{M} = [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!] = \sum_{r=1}^{R} \lambda_r \mathbf{a}_r^{(1)} \circ \cdots \circ \mathbf{a}_r^{(N)}$$

where $[\![\cdot]\!]$ is the compact notation for a CP model (also known as the Kruskal operator), $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_R]$ is a weight vector, and each $\mathbf{A}^{(n)}$ is an $I_n \times R$ *factor matrix* containing R vectors associated with dimension n in the data. The columns of these matrices are used in the outer products of the decomposition, where $\mathbf{A}^{(n)} = [\mathbf{a}_1^{(n)} \ldots \mathbf{a}_R^{(n)}]$.

In practice, a small value of R, relative to $I_1, \ldots, I_N$, is chosen to approximate $\mathcal{X}$ in CP form with a given *model*, $\mathcal{M}$. The model, $\mathcal{M}$, is determined based on assumptions of how the data $\mathcal{X}$ was generated. Maximum likelihood estimation is often used to fit these CP models, where a specific likelihood function associated with the data assumptions is used. The original methods for computing CP decompositions assumed Gaussian data and used a form of block coordinate descent optimization, which is now referred to as the CP-ALS, or alternating least squares, method [8], [9]. More recent work on decompositions for matrices and tensors with sparse count data assumes the data is Poisson generated and use the same alternating block optimization to fit the models under those assumptions. For example, Lee and Seung developed the nonnegative matrix factorization (NMF) algorithm assuming the underlying data is Poisson generated [10], and Chi and Kolda extended their work to CP decompositions of tensors under the same assumptions to develop the Canonical Polyadic Alternating Poisson Regression (CP-APR) method [2].

CP-APR is designed to minimize the negative log likelihood function of the Poisson distribution (up to an additive constant), $f(\cdot)$, given in matrix form as follows:

$$\min_{\mathbf{B}^{(n)} \geq 0} f(\mathbf{B}^{(n)}) = \mathbf{e}^T [\mathbf{B}^{(n)} \boldsymbol{\Pi}^{(n)} - \mathbf{X}_{(n)} * log(\mathbf{B}^{(n)} \boldsymbol{\Pi}^{(n)})]\mathbf{e} \quad (1)$$

where $\mathbf{e}$ is a vector of all ones, $\mathbf{B}^{(n)} = \mathbf{A}^{(n)} \boldsymbol{\Lambda}$, $\boldsymbol{\Lambda}$ is a diagonal matrix with $\lambda_1, \ldots, \lambda_N$ along the diagonal, $\mathbf{X}_{(n)}$ is an matricized version of $\mathcal{X}$ unfolded along dimension n, $*$ denotes element-wise matrix multiplication, and *log* indicates element-wise logarithm. As noted in [3], $f(\mathbf{B}^{(n)})$ in (1) can be written as a sum of completely separable convex row subproblems, each a function of R variables as follows:

$$f(\mathbf{B}^{(n)}) = \sum_{i=1}^{I_n} f_{\text{row}}(\mathbf{b}_i^{(n)}, \mathbf{x}_i^{(n)}, \boldsymbol{\Pi}^{(n)}), \quad (2)$$

where

$$f_{\text{row}}(\mathbf{b}_i^{(n)}, \mathbf{x}_i^{(n)}, \boldsymbol{\Pi}^{(n)}) = \sum_{r=1}^{R} b_r - \sum_{j=1}^{I_n} x_j \log \left(\sum_{r=1}^{R} b_r \pi_{rj} \right). \quad (3)$$

The form of the loss function in (2) is amenable to *parallelization* due to the separability of $f(\mathbf{B}^{(n)})$ into functions associated with row subproblems, f_{row}, and *second-order Newton-based optimization* can be used to solve each row subproblem due to the small number of variables, R, in (3).

The PDNR method for solving CP-APR is shown in Algorithm 1; see [3], the original publication of the PDNR method, for a detailed description of the complete algorithm. We focus here on lines 6–9, where the minimization of $f(\mathbf{B}^{(n)})$ is performed row-wise using (2) to obtain $\mathbf{b}_i^{(n)}$ for all $i = 1, \ldots, I_n$. For each row, the input data $\mathbf{x}_i^{(n)}$ is prepared from the unfolded tensor $\mathbf{X}_{(n)}$ (line 7) for the constrained Newton optimization solver (line 8).

The algorithm for minimizing each of the row subproblems in the PDNR method is shown in Algorithm 2. This minimizer is based on second-order damped Newton minimization, which requires the gradient (line 5) and Hessian (line 6) of $f_{row}(\cdot)$. To compute the damped Newton direction, a linear system is solved using a shifted Hessian matrix (shifted by a damping parameter μ_i to guarantee that the matrix is positive and semidefinite) and the negative gradient (line 8). Lastly, to satisfy the bound constraints, $\mathbf{b}_i^{(n)} \geq 0$, a line search is performed along the damped Newton direction and the result is projected onto the nonnegative orthant using the operator P_+. The line search is an iterative backtracking method that

Algorithm 1: CP-APR PDNR Algorithm [3]

1 CP-APR PDNR $(\mathcal{X}, \mathcal{M}, R)$;
Input : Sparse tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \ldots I_N$
Output: $\mathcal{M} = [\![\boldsymbol{\lambda}; \mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}]\!]$ with R components
2 **Initialize**
3 **repeat**
4 **for** $n = 1, \ldots, N$ **do**
5 $\boldsymbol{\Pi}^{(n)} = (\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \ldots \mathbf{A}^{(1)})^T$
6 **for** $i = 1, \ldots, I_n$ **do**
7 Extract row i from $\mathbf{X}_{(n)}$ as $\mathbf{x}_i^{(n)}$
8 Minimize $f_{\text{row}}(\mathbf{b}_i^{(n)}, \mathbf{x}_i^{(n)}, \boldsymbol{\Pi}^{(n)})$, subject to $\mathbf{b}_i^{(n)} \geq 0$, where $\mathbf{b}_i^{(n)}$ is row i of $\mathbf{B}^{(n)}$
9 **end**
10 $\boldsymbol{\lambda} = \mathbf{e}^T \mathbf{B}^{(n)}$ where $\mathbf{B}^{(n)} = [\mathbf{b}_1^{(n)} \ldots \mathbf{b}_{I_n}^{(n)}]^T$
11 $\mathbf{A}^{(n)} \leftarrow \mathbf{B}^{(n)} \boldsymbol{\Lambda}^{-1}$, where $\boldsymbol{\Lambda} = \text{diag}(\boldsymbol{\lambda})$
12 **end**
13 **until** *max iterations reached or convergence achieved*;

Algorithm 2: Row Subproblem Minimization

Row Subproblem Minimization $(\mathbf{x}_i^{(n)}, \mathbf{b}_i^{(n)})$;
Input : $\mathbf{x}_i^{(n)}$, row i of $X_{(n)}$ and current parameters $\mathbf{b}_i^{(n)}$
Output: $\hat{\mathbf{b}}_i^{(n)}$, updated parameters
Initialize
repeat
  Extract $\boldsymbol{\Pi}_i^{(n)}$ corresponding to nonzeros of $\mathbf{x}_i^{(n)}$
  Compute gradient $\mathbf{g}_{\mathrm{F}}^i = \nabla f_{\mathrm{row}}(\mathbf{b}_i^{(n)}, \mathbf{x}_i^{(n)}, \Pi_i^{(n)})$
  Compute Hessian $\mathbf{H}_{\mathrm{F}}^i = \nabla^2 f_{\mathrm{row}}(\mathbf{b}_i^{(n)}, \mathbf{x}_i^{(n)}, \boldsymbol{\Pi}_i^{(n)})$
  Solve: $\mathbf{d}_{\mathrm{F}}^i = -(\mathbf{H}_{\mathrm{F}}^i + \mu_i I)^{-1}\mathbf{g}_{\mathrm{F}}^i$
  Line search: $\hat{\mathbf{b}}_i^{(n)} = P_+[\mathbf{b}_i^{(n)} + \alpha \mathbf{d}_{\mathrm{F}}^i]$
until *max iterations reached or convergence achieved*;

involves evaluation of the objective function in (3) during each iteration to determine the step size, α.

B. Parallelizing Sparse CP Decomposition

Parallelization of sparse CP decomposition has been explored extensively on distributed memory [11]–[14], shared memory [13]–[16], and accelerator [17] systems. This work resulted in several performance enhancements, including data representations of sparse tensors that supports reduced computation, data distribution and partitioning schemes, and effective use of memory hierarchy such as cache and register blocking. However, the focus of much of this previous work is CP-ALS algorithms, which is suitable for Gaussian-generated data. For CP decompositions of Poisson-generated data, research on parallel sparse CP-APR algorithms has focused on reduced memory usage for shared memory implementations [5] and load balancing for distributed computing platforms [6]. To our knowledge, all previous work on CP-APR parallelization has focused on the method based on Multiplicative Updates [2] rather than on PDNR.

C. Kokkos and Sparse CP Decomposition

Kokkos [7] is a parallel programming framework designed for performance portability, leveraging modern C++, including templates, functors and lambda expressions. Kokkos supports two major abstractions, called **memory** and **execution spaces**. The former indicates data layout and memory location (device or host) with a light-weight C++ class (`Kokkos::View`) replacing arrays and `std::vector`. The latter indicates the location of parallel loop execution (`Kokkos::parallel_for`) and resource allocations and scheduling (`Kokkos::TeamPolicy`). These abstractions hide architecture and runtime-specific features, resulting in code similar to OpenMP programming. To achieve both code simplicity and portable performance, Kokkos provides a thin layer of application programming interfaces (APIs) and leverages the underlying runtime software of particular architectures (Pthread, OpenMP, CUDA, and HIP) to perform the management of memory allocations and parallel executions. Phipps and Kolda developed an efficient sparse CP-ALS with Kokkos [18]. We apply similar ideas for our parallel implementations of sparse CP-APR tensor decomposition algorithms.

III. SparTen and Parallel PDNR with Kokkos

SparTen is written with Kokkos to enable node-parallel sparse CP-APR for multiple platforms. Its design philosophy is similar to the parallel sparse CP-ALS code by Phipps and Kolda [18] including: (1) implementing data structures using `Kokkos::View` (Kokkos' portable data abstraction), (2) using coordinate format (COO) for sparse tensors, (3) applying `Kokkos::parallel_for` to perform data parallel linear algebra operations, and (4) leveraging atomic operations provided by Kokkos in a limited manner. Kokkos' `parallel_for` is readily applicable to the data parallel computation of matrix multiplication, such as for the Khatri-Rao product (denoted by $\odot$ on line 5 in Algorithm 1) or by the updates to progress towards a minimizer in the CP-APR Multiplicative Update method [2]. In contrast, the major benefit from parallelism in the PDNR method is concurrent minimization of independent row subproblems, which differs from classic data parallelism scenarios. In the remainder of this section we discuss the details of our parallel implementation of PDNR, including how Kokkos is leveraged and potential load imbalance is addressed.

A. Parallel PDNR with Kokkos

The implementation of PDNR in SparTen uses Kokkos' `parallel_for` to execute multiple row subproblem minimizations concurrently. As shown in Listing 1, Kokkos provides a way to control the amount of computing resources used by loop instances through the `Kokkos::TeamPolicy` class, which provides a three-level hierarchy of processing elements (`league`, `team` and `thread`). Parallelism in the inner loops is controlled by the execution policy for the `team`s of `thread`s (`Kokkos:TeamThreadRange`) or the vector lane of a `thread` (`Kokkos::ThreadVectorRange`). Our single-kernel *team-based* implementation assigns a single `team` to execute Row Subproblem Minimization from Algorithm 2. We use `Kokkos::TeamPolicy` to assign a single thread or single warp to a single team for CPUs and NVIDIA GPUs, respectively.

B. Parallel Row Subproblem Minimization with Kokkos

Row Subproblem Minimization in Algorithm 2 can be parallelized by adapting the data structure of the sparse tensor $\mathcal{X}$. Parallelization is applied over the nonzero entries of each row ($\mathbf{x}_i^{(n)}$) extracted from the unfolded sparse tensor $\mathbf{X}_{(n)}$. This unfolding does not alter the COO format of the tensor $\mathcal{X}$, but accessing the nonzero entries by rows involves extra indexing.[2]

Row Subproblem Minimization consists of three major computational components: (1) computation of the gradient and Hessian, (2) solution of the linear system involving the Hessian and gradient, and (3) projected line search. Each component has different parallelization requirements. The gradient and Hessian are computed from nonzero entries x_j of the sparse row $\mathbf{x}_i^{(n)}$. If the the number of nonzero entries is large,

[2] A similar idea is exploited for the parallel CP-ALS solver in [18] to reduce the atomic operations for the concurrent updates of the factor matrices.

```
typedef TeamPolicy<ExecutionSpace>::member_type member_type;
TeamPolicy<ExecutionSpace> policy( league_size, team_size, thread_size );

Kokkos::parallel_for( policy,  KOKKOS_LAMBDA( member_type team_member ) {
  int myRow = team_member.league_rank () * team_member.team_size ()
  + team_member.team_rank ();
  NewtonMethod( team, modeNumber, myRow, sparseTensor, factorMatrix);
  Kokkos::parallel_for( Kokkos::ThreadVectorRange(team, nComponents),[&] (const int i )} {
      // Computation
  }); });
```

Listing 1. `Kokkos::TeamPolicy` example.

the parallelization provides enough work for each processing element to perform concurrent updates to the gradient and Hessian (lines 5 and 6). We apply Kokkos' `ScatterView`, which creates a local copy of each team or thread to handle concurrent contributions from multiple teams or threads to reduce the number of atomic memory accesses. The solution of the linear system can be parallelized, but it is not effective for the relatively small problem sizes (R) we use in PDNR; performance tuning for decompositions with larger values of R is left for future work. The projected line search involves small vector operations (of size R) to evaluate the solution ($\hat{\mathbf{b}}_i^{(n)}$), and objective function evaluations involve a `parallel_reduce` operation. The objective function computation can be as expensive as computing the gradient and Hessian, as it requires access to all nonzero entries in $\mathbf{x}_i^{(n)}$. Our single-kernel team-based implementation uses `ThreadVectorRange` with `team`, dispatched by the outermost `parallel_for`.

C. Addressing Load Imbalance in PDNR

By default, resource management using Kokkos largely depends on the underlying runtime implementation. The current Kokkos design assumes that individual leagues, teams, and threads perform the same amount of work for efficient parallelization. This situation is ideal if the number of nonzero entries in each row subproblem (i.e., $\mathbf{x}_i^{(n)}$ in $\mathbf{X}_{(n)}$) is the same, but this is unlikely for most sparse tensor problems from real applications. For example, Table I lists sparse tensors from count data available in the FROSTT collection [19], indicating the poor aspect ratios of the sizes of the modes for the *chicago* and *lbnl* tensors. A closer look reveals that some tensors exhibit an extreme variation of nonzero entry distribution over the rows in a few modes. For example, Table II lists statistics of the nonzero entry distribution in *lbnl* from the unfolded tensor associated with each mode. We note that a single row in the first mode accounts for approximately 25% (400,000) of nonzero entries, while the average number of nonzeros per row in this mode is approximately 1,000. On the other hand, the fifth mode of the same tensor exhibits a flat distribution; the largest nonzero count is only 13 and the average count is 1.95 with a small standard deviation of 0.86.

This extreme load imbalance indicates a potential performance bottleneck with our single-kernel team-based work assignment. Unless the underlying hardware has the ability to control the frequency of individual core usage and data traffic among cores, the execution time for solving the largest row subproblem will be a limiting scaling factor. Modern multicore CPUs have some voltage control to adaptively tune the performance of individual cores, and their memory controllers can balance the data traffic to individual cores. However, accelerators (GPUs) are not designed to handle such load imbalance among cores (and groups of cores).

A possible approach to mitigate this load imbalance is assigning either more `team`s to row subproblems with large numbers of nonzero entries over those with relatively few nonzero entries or variable computing resources to individual `team`s. However, neither of these approaches is feasible using the current version of Kokkos. Under the same `TeamPolicy`, all resource abstractions in the same level of the hierarchy retain the same amount of computing resources; there are no APIs for users to manually schedule individual abstractions. For example, thread synchronization is allowed within either a single `team` (`team_barrier`) or a single node (`fence`), but no synchronization for the threads over specified teams. This partly reflects current limitations of accelerator architectures; for example, CUDA threads cannot synchronize with those from external thread blocks.

To overcome these resource scheduling restrictions, we have developed a special *all-node* row subproblem kernel that assigns the computing resources of an entire node to a single row subproblem when the number of nonzero entries is larger

TABLE I
TENSORS FROM THE FROSTT COLLECTION [19]

FROSTT Name (short name)	Mode sizes
chicago_crime_comm (chicago)	$(6186, 24, 77, 32)$
delicious4d (delicious)	$(532924, 17262471, 2480308, 1443)$
lbnl_network (lbnl)	$(1605, 4198, 1631, 4209, 868131)$
nell_2 (nell)	$(12092, 9184, 28818)$

TABLE II
DISTRIBUTION OF NONZERO ENTRIES PER ROW OF *lbnl* TENSOR

Mode	Average	Min	Max	Std Dev
1	1058.46	1	397413	11906.86
2	404.67	1	768321	12242.96
3	11041.59	1	310733	11183.01
4	403.62	1	676548	11906.86
5	1.96	1	13	0.86

than a user-defined threshold value. The all-node kernel is parallelized from the outermost `parallel_for` over nonzero entries for the computation of the gradient, Hessian, and objective function, whereas the single-kernel implementation calls the team-based kernel, as indicated in Listing 1.

The PDNR implementation is modified to execute a *multi-kernel* approach that combines team-based and all-node kernels based on the number of nonzeros, and the choice of which kernel to use is determined using simple thresholds. For rows with large nonzero count, the all-node kernel is used; and for the remaining rows with fewer nonzeros, the team-based kernel from the single-kernel approach is used.

IV. Parallel PDNR Performance

In this section, we present the preliminary performance evaluations of our multi-kernel approach for solving CP-APR tensor decomposition using the parallel PDNR method. The code is written in C++ using Kokkos version 2.9 for parallelization. We present performance results on three different computing platforms (sockets×cores, clock speed, memory):

- **Marvell ARM ThunderX2**: 2×28, 2.0 GHz, 128 GB;
- **Intel Xeon Broadwell**: 2×18, 2.1 GHz, 128 GB; and
- **NVIDIA Volta V100**: 5120, 1.5 GHz, 32 GB.

Kokkos uses OpenMP for the CPU systems and CUDA 10.0 for this GPU system. The test program is built with gcc 8.3.1, gcc 7.2.0, and nvcc 10.0 (gcc 7.3.0 for the host CPU) on the Broadwell, ThunderX2 and V100 systems, respectively. The compiler flags are set to those provided by the Kokkos' build system (`-O3` and architecture specific optimization options). The data used in all experiments are listed in Table I; and we use short names in our descriptions/plots below.

Because our focus is on runtime performance, we limit PDNR to 10 outer iterations to compute rank-10 decompositions. In each outer iteration, we run the Row Subproblem Minimization up to 10 iterations per row subproblem using up to 10 back-tracking steps for each line search. Unified memory is disabled on the GPU, so data traffic between the CPU and GPU is managed by Kokkos' `deep_copy` calls to keep all data structures (sparse tensor, factor matrix, and temporary storage, including $\boldsymbol{\Pi}$, gradient and Hessian) in the GPU memory. We report the execution time of the PDNR solver only; other runtime operations, such as loading data from files and moving data to GPUs, are not reported here.

A. Scalability of Single-Kernel Parallel PDNR

Strong scalability results as a function of the number of OpenMP threads for the single-kernel parallel PDNR implementation on the CPU systems is presented in Figure 2. The general trend for all data tensors across both the ThunderX2 and Broadwell platforms indicated that using more threads, include hyperthreads, leads to improved performance. The performance changes from 8 to 14 cores on ThunderX2 and 8 to 16 cores on Broadwell is the only exception, and determining the cause of these changes are left as future work. On ThunderX2, we see that moving from 14 to 28 threads (i.e., fully saturating a single socket's cores) and moving from

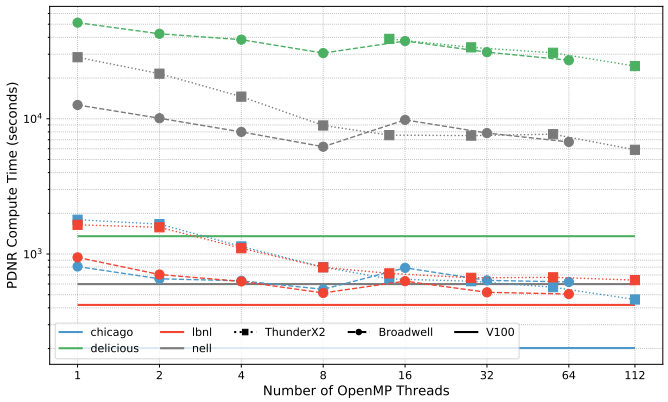

Fig. 2. Single-kernel strong scalability results.

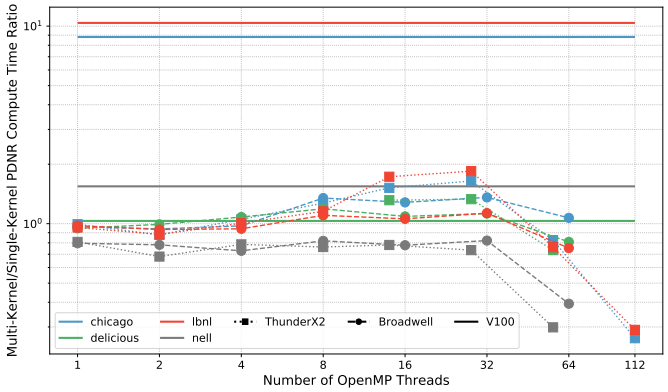

Fig. 3. Multi-kernel versus single-kernel scalability, switch threshold = 2^{14}.

28 to 56 threads (i.e., moving from using cores on a single socket to cores on both sockets) leads to little improvement. However, the use of hyperthreading on ThunderX2 does lead to improved performance on all of the data tensors. We do not see this same performance behavior on Broadwell when saturating socket and core usage, as the general trend on that platform is increased performance with more threads and hyperthreads. Performance on the V100 GPU platform is depicted as horizontal lines for the different data tensors. As expected, the absolute compute times are much lower for V100 than for ThunderX2 and Broadwell.

B. Improved Scalability of Multi-Kernel Parallel PDNR

To demonstrate the effectiveness of the multi-kernel approach, we conduct experiments using several threshold values for switching between the team-based and the all-node kernels. Figure 3 presents a comparison of scalability results for the multi-kernel versus single-kernel approach, showing the ratio of strong scalability as a function of the number of OpenMP threads used on the CPUs, with the corresponding results on the GPU depicted as horizontal lines. The results in the figure use a single switch threshold value of 2^{14} for the multi-kernel approach. On the GPU, there is a benefit in using the multi-kernel approach on all data tensors, with the

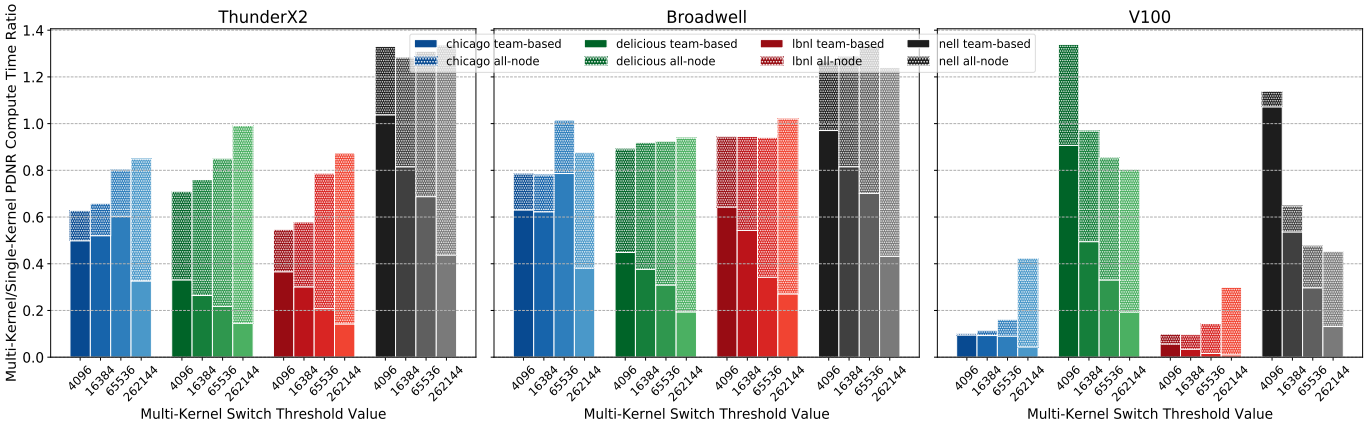

Fig. 4. Multi-kernel versus single-kernel performance by kernel type as a function of the switch threshold value used in the multi-kernel approach.

most dramatic benefit—more than a 10-fold improvement—occurring on the smaller *lbnl* tensor. On ThunderX2, we see improved performance until we saturate core usage on a single socket (i.e., 28 cores), and then performance degrades when moving to both sockets and hyperthreading. On Broadwell, though, moving from single to dual socket usage (i.e., 16 to 32 cores) still leads to improvement, whereas using hyperthreads leads to degraded performance. Note that performance on CPUs for the *nell* data is always better using the single-kernel approach; understanding this behavior is left as future work.

Figure 4 shows the relative performance improvement of the multi-kernel approach over the single-kernel approach, where CPU results are those with the best overall performing thread count (i.e., 14 threads on ThunderX2 and 16 threads on Broadwell). These results illustrate the relative time spent in team-based (solid bars) and all-node (shaded bars) kernels for the multi-kernel approach, with the total height of the bars depicting the ratio of the multi-kernel versus single-kernel PDNR compute times. We observe that 1) the selection of the threshold value is critical for optimizing the performance; 2) the multi-kernel approach is very effective on GPUs and somewhat effective on CPUs; and 3) the multi-kernel approach is more effective for tensors with extremely small mode sizes (*chicago*) or a high degree of imbalance in the nonzero distribution (*lbnl*). Overall, we see significant performance improvements when using the multi-kernel approach versus the single-kernel approach—up to 80% on CPUs (ThunderX2) and over 10-fold on GPUs. Again we see that the multi-kernel approach is not effective for the *nell* tensor data on CPUs, this time across all switch threshold values.

V. Conclusions

We have presented two parallel implementations of the PDNR method for CP-APR tensor decompositions, focusing on concurrent execution of damped Newton optimization for row subproblems. We presented a single-kernel approach that leverages Kokkos' `parallel_for` and `TeamPolicy` to map `team` to row subproblems on a one-to-one basis. The approach is designed to leverage Kokkos' nested parallelization to achieve improved performance over serial implementations of PDNR. However, load imbalance due to irregular mode sizes and nonzero entries per row, as seen in real-world tensor problems, leads to non-optimal performance. We mitigate this problem with a multi-kernel approach that adaptively selects the resource mapping to the row subproblem instances. Our preliminary results indicate that a simple two-kernel approach is effective on GPUs, achieving 10-fold performance improvement. With the recent multicore CPUs, our multi-kernel approach still improves the performance for the tensors that exhibit extreme irregularity in the mode size and nonzero distribution, but the original single-kernel approach still outperforms this approach in problems that do not exhibit these irregular patterns.

Future work includes development of row subproblem minimization kernels that can run on multiple (but not all) teams to bridge the gap between the team-based and all-node kernels presented here. However, this would be an advanced use of Kokkos that has not yet been demonstrated in practice and thus could present unforeseen research challenges. Another research direction is tuning the memory profile of parallel PDNR to better understand the performance tradeoffs in the multi-kernel approach. Finally, the current implementation in SparTen of parallel PDNR computes $\boldsymbol{\Pi}$ on the fly to reduce the memory requirements within each kernel launch. An alternative approach would be to dynamically manage the tradeoffs between the storage for $\boldsymbol{\Pi}$, concurrency (number of nonzero entries being computed), and the cost of on-fly computation. This idea could be compared to the memory-saving CP-APR technique proposed by Baskaran [5] which has also shown promise for the Multiplicative Update method.

Acknowledgement

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

REFERENCES

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

[2] 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. [Online]. Available: https://doi.org/10.1137/110859063

[3] 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, Apr. 2015.

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

[5] M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin, "Memory-efficient parallel tensor decompositions," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[6] M. Baskaran, T. Henretty, and J. Ezick, "Fast and scalable distributed tensor decompositions," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[7] 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*, vol. 74, no. 12, pp. 3202–3216, 2014.

[8] J. D. Carroll and J. J. Chang, "Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition," *Psychometrika*, vol. 35, pp. 283–319, 1970.

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

[10] D. D. Lee and H. S. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, no. 6755, pp. 788—791, 1999.

[11] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–11.

[12] 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)*, May 2017, pp. 1058–1067.

[13] M. Baskaran, B. Meister, R. Lethin, and J. Cai, "Optimization of symmetric tensor computations," in *IEEE Conference on High Performance Extreme Computing (HPEC), Waltham, MA, USA*. IEEE, Sep. 2015.

[14] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '15. USA: IEEE Computer Society, 2015, p. 61–70. [Online]. Available: https://doi.org/10.1109/IPDPS.2015.27

[15] B. Allan, "Optimization of CPAPR for x64 multicore," Sandia National Laboratories, Tech. Rep. SAND2011-9432, Dec. 2011.

[16] S. Smith, J. Park, and G. Karypis, "An exploration of optimization algorithms for high performance tensor completion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 31:1–31:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014946

[17] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 47–57.

[18] 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.

[19] 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/