Performance-Portable Sparse Tensor Decomposition Kernels on Emerging Parallel Architectures

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

Keita Teranishi
Sandia National Laboratories
knteran@sandia.gov

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

Jee Choi
University of Oregon
jeec@uoregon.edu

***Abstract*—We leverage the Kokkos library to study performance portability of parallel sparse tensor decompositions on CPU and GPU architectures. 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.**

I. Introduction

Many real-world data analysis applications—e.g., in healthcare, cybersecurity, social networks, and more—give rise to multi-way data that can be naturally represented by sparse tensors. 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].

One of the key performance bottlenecks in tensor decomposition is the matricized tensor times Khatri-Rao product (MTTKRP) found in the CANDECOMP/PARAFAC Alternating Least Squares (CP-ALS) algorithm for the canonical polyadic decomposition (CPD) model, which approximates a tensor as a sum of K rank-one tensors [1].

Here we explore the performance of *parallel tensor decomposition* by implementing a set of proxy benchmark kernels using the Kokkos C++ performance-portable library [2]. With the emergence of drastically different parallel architectures, performance portability is critical in achieving optimal productivity on heterogeneous computing systems.

We start with simple operations from the STREAM benchmark [3] and then extend the evaluation to the full MTTKRP kernel. We choose STREAM for two reasons: (i) MTTKRP on sparse tensors is bandwidth-bound, as are the STREAM operations, and (ii) STREAM operations can be used as building blocks for MTTKRP, as described in §II.

Contributions: We make three key contributions towards performance portable sparse MTTKRP:

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.

II. Methods

We first describe the methods we use in evaluating parallel performance. Although generally applicable, we limit discussion to tensors with three dimensions for simplicity.

A. CP-ALS and MTTKRP

Given a 3-way tensor $\mathcal{X}$ of size $I_1 \times I_2 \times I_3$, the CP-ALS algorithm computes a rank-K model tensor $\mathcal{M}$, consisting of factor matrices $A \in \mathbb{R}^{I_1 \times K}$, $B \in \mathbb{R}^{I_2 \times K}$, and $C \in \mathbb{R}^{I_3 \times K}$, 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):

$$T(:) \leftarrow B(j,:) * C(k,:) \quad \text{element-wise product} \quad (1)$$

$$T(:) \leftarrow v * T(:) \quad \text{scale} \quad (2)$$

$$A(i,:) \leftarrow A(i,:) + T(:) \quad \text{element-wise add} \quad (3)$$

where $A(i,:)$, $B(j,:)$ and $C(k,:)$ correspond to the rows of the factor matrices. This is repeated for every non-zero element.

B. Challenges

We can see from the above equations that MTTKRP exhibits *low arithmetic intensity* (i.e., it is memory bandwidth-bound). Additionally, the last step (Equation 3) introduces 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 $A(i,:)$ need to be serialized.

However, we can also see that these operations are similar to those found in the STREAM benchmark. Therefore, we use the STREAM benchmark as a proxy for the MTTKRP kernel, and the MTTKRP kernel—taken from the Parallel Sparse Tensor Algorithm Benchmark Suite (PASTA) library [4]—as a proxy for the full CP-ALS algorithm.

C. 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. (Note that OpenMP 4.5+ supports offloading to GPU devices [5], but

we use Kokkos for performance portability due to its ability to efficiently handle data layout for both dense and sparse operations.) 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.

III. Experimental Results

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

A. Test Systems and Data

We evaluate our kernels on the nine systems shown in the left table below. For the kernels in the STREAM benchmark, we use up to $500M$ elements per array. For the MTTKRP kernel, we use the real-world tensors from the FROSTT [6] website shown in the right table below.

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

Test Systems

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

Test Data

B. Analysis

Figure 1 shows the achieved bandwidth (bars) from various STREAM operations and the speedup (line) 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 2 shows the same for the MTTKRP benchmark. For MTTKRP, we achieve *superior* performance on CPUs. For GPUs, our Kokkos enhanced kernel achieves 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 lack of hardware atomic operations for double-precision data.

IV. Discussion and Future Work

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 expensive atomic operations.

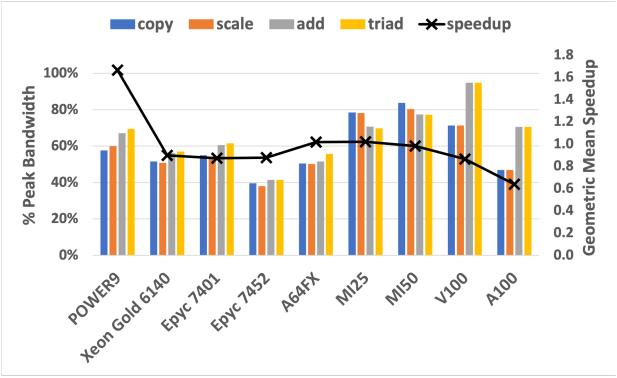

Fig. 1: Performance results on the STREAM benchmarks.

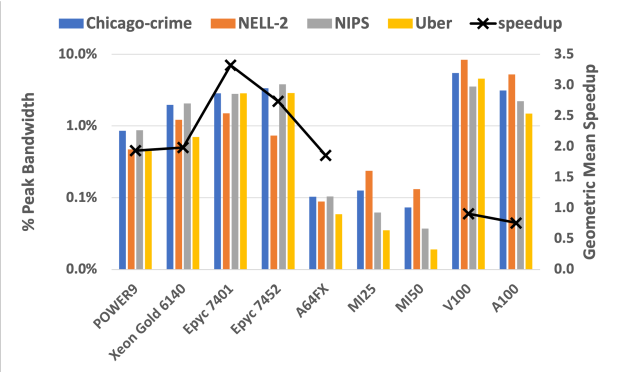

Fig. 2: Performance results on the MTTKRP benchmark.

Acknowledgment

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-NA-0003525.

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.

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

[3] 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.

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

[5] 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.

[6] 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/