2022 Future Computing Summer Internship:
TenTS (Tensor Toolbox in Scala)

Nathan Nguyen (nathan.tm.nguyen@tamu.edu)
Luke Vandecastelee (lvandec@uoregon.edu)
Teresa Ranadive (tranadive@lps.umd.edu)
Thomas Rolinger (tbrolin@lps.umd.edu)

November 16, 2022

Abstract

Tensor decomposition algorithms are a collection of recent methods utilized for big data exploration that frequently make use of high performance computing (HPC) systems. However, implementing and parallelizing such algorithms often requires specialization to specific hardware, in order to be efficient on real-world applications. This specialization requires significant overhead, incurred when porting tensor algorithms to new hardware. As such, tensor decomposition has been slow to gain widespread prominence as a data exploration tool. Therefore, this work investigates several new implementations of tensor decomposition methods, with the goal of increasing portability, while maintaining scalability. In particular, we present an implementation of the CANDECOMP/PARAFAC-Alternating Poisson Regression (CP-APR) decomposition algorithm in Scala. We also utilize the novel Adaptive Linearized Tensor Order (ALTO) sparse tensor storage format and leverage Apache Spark for increased parallelism across multiple nodes in a cluster. Through our research and experiments, we found that there exist inherent restrictions within Scala that severely reduce algorithmic performance. For instance, certain performance difficulties arise when using Scala in conjunction with the ALTO tensor format. Furthermore, we found limitations within both Scala and Spark that narrowed the range of tensors that could be decomposed. However, a multi-threaded Scala implementation aimed at reducing these limitations yielded up to a 12x speedup for both ALTO and other formats.

1 Introduction and Motivation

Over the past decade, tensor decomposition has gained substantial interest as a big data exploration method [1, 2, 3]. Tensor decomposition algorithms can extract meaningful patterns from large, sparse data sets, without any prior information on the underlying nature of the data. Applications for these algorithms span a wide variety of fields including cyber-security [1], geospatial analysis [2, 3], text analysis [1], signal processing, computer vision, data mining, neuroscience, graph analysis and others [1]. With a diverse application range, it is necessary to create tensor decomposition software with portability in mind. Tensor algorithms frequently make use of high performance computing (HPC) systems to efficiently compute decompositions. Consequently, it is imperative to consider the target HPC system architecture when designing tensor algorithms. Although several tensor algorithm implementations [1, 2] make this consideration, portable tensor software is notoriously difficult to implement, due to vast differences in hardware across HPC systems. If tensor software can be made more portable, tensor decomposition algorithms would likely grow as a tool for many applications.

Our research aims to extend the range of target HPC systems for tensor software by implementing the tensor decomposition algorithm CANDECOMP/PARAFAC-Alternating Poisson Regression (CP-APR) [1] in the Scala programming language. Additionally, we leverage the Apache Spark API to scale this algorithm across distributed-memory systems. The Scala programming language and Apache Spark API have been used to design portable frameworks for both machine learning and data science applications. This work investigates how these tools might be utilized to design portable tensor decomposition algorithms.
Due to the size and sparsity of large data sets, the choice of tensor storage format can significantly impact performance. As a result, our CP-APR implementation utilizes the novel sparse tensor storage format Adaptive Linearized Tensor Order (ALTO) \[1\]. ALTO is a compressed storage format that improves algorithmic performance, via data movement reduction, for serial and parallel tensor decomposition algorithms, while simultaneously reducing overall memory usage.

We therefore make the following research contributions with this work:

1. We implement CP-APR, using the ALTO storage format, in the Scala programming language, utilizing Apache Spark to parallelize CP-APR for distributed memory systems.

2. We investigate methods for optimizing the CP-APR Scala implementation to improve performance, as well as a solution for reducing Java Virtual Machine (JVM) restrictions in order to accommodate larger sparse tensors.

3. We discuss several problems that arise when combining Scala with ALTO. This includes an investigation into the optimizations made by the Just-In-Time compiler (JIT).

4. We implement a multi-threaded version of CP-APR, using ALTO, in response to several Apache Spark performance limitations that emerge with computationally heavy workloads.

5. We evaluate the performance of all of the aforementioned implementations of CP-APR. These implementations all yield varying forms of positive scaling across an increasing number of threads. Using up to 32 threads, we observe speedups as large as 7x for Apache Spark, and 14x for a manually threaded ALTO implementation. These results verify the performance gains obtained by ALTO \[1\], when utilized for sparse tensor decomposition.

The rest of this report is organized as follows. Several relevant background topics including tensors and tensor decomposition, ALTO, Scala, and Apache Spark are introduced in Section 2. We discuss our approach to designing and optimizing various CP-APR Scala implementations in Section 3. Next, a performance evaluation of our CP-APR implementations is presented in Section 4. Finally, concluding remarks and a discussion of future work is given in Section 5.

2  Background and Prior Work

In this section, we discuss several relevant background topics, including tensors and tensor decomposition (cf. Section 2.1), ALTO (cf. Section 2.2), Scala (cf. Section 2.3), and Apache Spark (cf. Section 2.4).

2.1  Tensors and Tensor Decomposition

Below, we introduce the concepts of tensors and tensor decomposition. For a more complete introduction, see \[1\].

For any positive integer \(N\), an \(N\)-mode or order \(N\) tensor \(\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}\) is a multi-modal array. For example, a vector is a tensor of order one, and a matrix is a tensor of order two. Since linear algebra addresses the study of matrices and vectors, tensor analysis usually focuses on tensors of order three and larger. Tensors can be used to represent real-world data sets with multiple attributes, due to their multi-modal structure. For example, in Figure 1, we see how the email metadata in the table on the left can be represented as the tensor on the right. Moreover, the number \(n\) of emails sent from sender A to recipient B on day C is indicated in the tensor by setting the \((A,B,C)\)-th entry of the tensor to \(n\), e.g., if Nathan sends Luke three emails on Monday, a three is placed in the tensor entry indexed by \((\text{Sender}, \text{Recipient}, \text{Day}) = (\text{Nathan}, \text{Luke}, \text{Monday})\).

For the purpose of applications, a rank-\(R\) CANDECOMP/PARAFAC (CP) tensor decomposition \(\mathcal{M}\) of an \(N\)-mode tensor \(\mathcal{X}\) is given by the sum of \(R\) outer products of \(N\) appropriately sized vectors, where \(\mathcal{M}\) is itself also an \(N\)-mode tensor that, in some sense, approximates \(\mathcal{X}\) (cf. Figure 2). Each outer product in this sum is also called a component of \(\mathcal{M}\). Components often correspond to interpretable behaviors that can be found in the data used to form a tensor. For example, we considered a tensor formed from NetFlow data collected over a period of 58 days and published by Los Alamos National Laboratories \[1\].
Figure 1: The email metadata in the table on the left can be represented as the tensor on the right: if sender A sends recipient B n emails on day C, the \((A, B, C)\)-th entry of the tensor is \(n\). For example, Nathan sends Luke three emails on Monday, so a three (circled) is placed in the tensor entry indexed by \((\text{Sender}, \text{Recipient}, \text{Day}) = (\text{Nathan}, \text{Luke}, \text{Monday})\).

Figure 2: A rank-\(R\) decomposition \(\mathcal{M}\) of a three-mode tensor \(\mathcal{X}\).

This tensor has modes corresponding to the attributes timestamp (binned into ten minute intervals), source device, destination device, destination port, and number of bytes transferred (binned in logarithmic scale). Performing a rank-100 decomposition of this tensor yielded the component displayed in Figure 3. Note that to obtain the plots in Figure 3, we first scale the vectors whose outer product forms this component to have a 1-norm of one, and then plot the entries of each scaled vector in a bar plot. The product of the scalars factored out of the vectors is called the weight of the component, and is displayed at the top of the figure. This weight (relative to the weights of other components) gives us an approximate idea of how much NetFlow traffic in the original data set can be described by this component.

The component in Figure 3 corresponds to a port scanning behavior. We can tell this behavior is a port scan, as it involves one machine pinging another, over a very large number of ports.

**CP–Alternating Poisson Regression (CP–APR)** is a tensor decomposition algorithm that is most appropriate for count tensors, i.e., tensors whose entries are all non-negative integers \([\cdot]\). Compared to the more commonly used tensor decomposition algorithm, CP–ALS (CP–Alternating Least Squares Regression) \([\cdot]\), CP–APR provides more interesting decompositions for count data in real-world applications. A sketch of the CP–APR algorithm is displayed in Figure 4.

### 2.2 Sparse Tensor Storage Formats and ALTO

The Coordinate (COO) format is the traditional sparse tensor storage format used by many libraries such as Tensor Toolbox \([\cdot]\), TensorFlow \([\cdot]\), and TensorLab \([\cdot]\), where the coordinates of the non-zeros in a sparse tensor are stored in one array, while the corresponding non-zero values are stored in another. The primary reason for widespread usage of COO is its simplicity, as well as the fact that it is a mode agnostic format (as opposed to a mode-specific format, where multiple copies of a tensor are stored, with each copy ordered to optimize data movement for tensor operations along a specific tensor mode). COO allows for a straightforward implementation of tensor algorithms in code. Additionally, since COO is a mode-agnostic
storage format, it requires less memory to store a given sparse tensor than that required by any mode-specific storage format. Despite the simplicity and widespread adoption of COO, using this tensor storage format can result in poor performance when computing decompositions of sparse tensors. This poor performance is a result of storing the tensor entries in an order that typically (i) does not minimize data movement and (ii) incurs a high rate of cache misses. Furthermore, when running parallel tensor algorithms, the COO format can result in a high synchronization overhead to resolve write/update conflicts across threads of execution, thus reducing overall execution time. Due to the performance limitations of COO, it is imperative to select an alternative sparse tensor storage format.

Adaptive Linearized Tensor Order (ALTO) is a novel compressed storage format that we utilized for our software. ALTO reduces many of the shortcomings of COO by (i) storing the non-zeros (and their corresponding indices) of a sparse tensor in memory in an order such that data movement and cache misses will be minimized for tensor operations across all modes, and (ii) reducing the memory footprint of the sparse tensor by compressing the set of indices for a given non-zero into a single index (cf. Figure 5). Moreover, this conversion from COO indices to ALTO indices improves cache locality, reduces overall memory usage, and reduces the number of synchronization conflicts for parallel tensor decomposition. Utilizing the ALTO sparse tensor storage format results in (i) significant speedup for key tensor kernels over all other state-of-the-art mode-agnostic storage formats and (ii) a reduced memory requirement relative to all state-of-the-art mode-specific formats.

An ALTO index consists of a single 64 or 128-bit integer value, depending upon the size of the tensor. ALTO compresses a set of COO indices to an ALTO index via the parallel bit deposit (PDEP) function, and extracts the COO indices from an ALTO index via the parallel bit extract (PEXT) function. Both PDEP and PEXT are a part of the Bit Manipulation Instruction set (BMI2) found in x86-64 assembly. Hence, both index compression and index extraction can be completed with a single assembly instruction, reducing the overhead of using the ALTO format. Figure 5 illustrates how ALTO performs index compression for a three-mode tensor.
Let $\mathbf{X}$ be a tensor of size $I_1 \times \cdots \times I_N$. Let $\mathbf{M} = (\lambda, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)})$ be an initial guess for an $R$-component model such that $\mathbf{M} \in \Omega(\zeta)$ for some $\zeta > 0$.

Choose the following parameters:
- $k_{\text{max}}$ = Maximum number of outer iterations
- $t_{\text{max}}$ = Maximum number of inner iterations (per outer iteration)
- $\tau$ = Convergence tolerance on KKT conditions (e.g., $10^{-4}$)
- $\kappa = $ Inadmissible zero avoidance adjustment (e.g., 0.01)
- $\kappa_{\text{tol}} = $ Tolerance for identifying a potential inadmissible zero (e.g., $10^{-6}$)
- $\epsilon = $ Minimum divisor to prevent divide-by-zero (e.g., $10^{-10}$)

1: for $k = 1, 2, \ldots, k_{\text{max}}$ do
2: $\text{isConverged} \leftarrow \text{true}$
3: for $n = 1, \ldots, N$ do
4: $s(n, r) \leftarrow \begin{cases} 1, & \text{if } k > 1 \text{ and } A^{(n)}(i, r) < k_{\text{tol}}, \text{ and } \Phi^{(n)}(i, r) > 1, \\ 0, & \text{otherwise} \end{cases}$
5: $B \leftarrow (A^{(n)} + S)A$
6: $\Pi \leftarrow \left( A^{(N)} \otimes \cdots \otimes A^{(n+1)} \otimes A^{(n-1)} \otimes \cdots \otimes A^{(1)} \right)^T$
7: for $\ell = 1, 2, \ldots, t_{\text{max}}$ do
8: $\Phi^{(n)} = (X_{\text{opt}} \otimes (\max(B) \cdot I)) \Pi^{\dagger}$
9: if $\min(B, \mathbf{E} - \Phi^{(n)}) < \tau$ then
10: break
11: end if
12: $\text{isConverged} \leftarrow \text{false}$
13: $B \leftarrow B \cdot \Phi^{(n)}$
14: end for
15: $\lambda \leftarrow c^T B$
16: $A^{(n)} \leftarrow B A^{-1}$
17: end for
18: if $\text{isConverged} = \text{true}$ then
19: break
20: end if
21: end for

Figure 4: Complete CP-APR algorithm

2.3 Scala

We chose to implement CP-APR in Scala, primarily due to potential customer interest. Scala is a functional language that targets the Java Virtual Machine (JVM). It features an expressive type system and extensive syntax sugar for an ergonomic developer experience. Since the Scala compiler emits JVM bytecode, Scala applications have first-class interoperability with existing libraries and tools intended for Java applications. Furthermore, Scala applications follow the same execution model as Java applications; the Scala compiler emits relatively unoptimized bytecode and defers to the JVM’s Just-In-Time compiler (JIT) to lower the bytecode to native code at runtime for maximum performance. Unfortunately, the use of Scala’s language abstractions can result in inefficient bytecode generation, which can inhibit the JIT’s ability to produce efficient code. We will discuss these limitations further in Section 3.

2.4 Apache Spark

Also due to potential custom interest, we chose to leverage Apache Spark [40] to handle parallelization within our implementation of CP-APR. Apache Spark is a data analytics engine in Scala used for exploratory data analysis and machine learning in the commercial industry. Spark attempts to deliver effortless scalability by abstracting away architectural details, enabling developers to make minimal modifications to sequential code to run it on a large, distributed scale. In particular, Spark takes care of partitioning the initial data set, dispatching data and tasks to workers, and dynamically adjusting the workload to best utilize the system’s resources in a fault tolerant fashion. For example, Figure 6 illustrates how Spark’s parallelize call distributes the range of that call to perform a Monte Carlo simulation used to approximate $\pi$. This is done without requiring the developer to specify how various worker threads and hardware devices should interact with the data. Although this abstraction reduces the demands on the developer and enhances
code readability, we will see in Section 3 that the high-level Spark API can present several challenges when finer-grained control over worker communication is required.

```scala
def approximatePi(sc: sparkContext, numPoints: Int): Double = {
  val count = sc.parallelize(1 until numPoints).map { i =>
    val x = Math.random * 2 - 1
    val y = Math.random * 2 - 1
    if (x * x + y * y <= 1) 1 else 0
  }.reduce(_ + _)
  4.0 * count / (numPoints - 1)
}
```

**Figure 6:** Monte Carlo simulation to approximate π in Apache Spark: the parallelize call lifts the range into a Resilient Distributed Dataset (RDD), which provides several methods for applying transformations and reductions on the data set.

## 3 Approach and Methodology

In this section, we will discuss several different implementations of CP-APR in Scala. We will first consider a single-threaded implementation of CP-APR in Scala in Section 3.1, along with several related optimizations in Sections 3.2–3.3. A analysis of some of the shortcomings of implementing ALTO in conjunction with Scala will follow in Section 3.4. Finally, in Section 3.5, we will discuss incorporating Apache Spark into our serial implementation for parallel distribution, along with the shortcomings of using Spark in HPC environments for computationally intense workloads.

### 3.1 Initial Implementation

Our first implementation of CP-APR was a direct port of Kolda's serial MATLAB implementation of the Tensor Toolbox [1] using COO. To perform matrix operations, we chose to leverage Breeze, a numerical processing library in Scala, since Spark uses Breeze internally for machine learning kernels and recommends its usage. While the resulting code was readable and easy to verify, it ran roughly 4x slower than the MATLAB
implementation on a small data set with approximately 5,000 non-zeros. It was extremely surprising to see that Scala, a compiled language with a JIT, was outperformed by MATLAB, an interpreted language. We observed similar disparities in performance for larger data sets and concluded that our single-threaded COO implementation would need significant optimization to be competitive with the MATLAB implementation. In the next two sections, we will discuss our efforts to optimize our initial implementation.

3.2 Optimizing with Ghidra and JProfiler

We applied both static and dynamic analysis techniques to identify and fix performance bugs within our Scala implementation. To perform static analysis of the JVM bytecode generated by the Scala compiler, we leveraged Ghidra\(^1\), a popular and open-source software reverse-engineering framework. One of Ghidra's primary features is decompilation, the process of transforming machine code into a human-readable, pseudo-C format. While there already exists tooling for disassembling machine code, it can be difficult to reason about the behavior of a program at the instruction level. Decompilation strikes a balance between preserving the original intent of the program and enabling the user to make productive judgments about the behavior of the program. While decompilation is primarily used for reverse-engineering, it can yield insightful information about how a compiler applies transformations to the initial source code. Using Ghidra's ability to decompile JVM bytecode, we were able to identify key performance issues caused by hidden allocations introduced by Scala's syntax sugar. For example, Figure 7 illustrates how a simple `for` loop (pictured on the left) performs multiple heap allocations, potentially inhibiting function inlining (pictured in decompiled code on the right). In particular, the iteration bounds of the loop are placed on the heap, which causes an indirection each iteration. Furthermore, the entire body of the loop is lifted into a closure, which incurs an additional jump on every iteration. While the JIT can partially mitigate the performance overhead of how `for` loops are desugared, we observed a non-negligible slowdown by replacing all `for` loops with `while` loops, which do not suffer from the same performance characteristics. Since Ghidra also has the ability to display the control flow graph of a program, we used this capability extensively to find unnecessary conditional branches within our implementation.

```
def f() = {
  for (i <- 0 until 10) {
    println(i)
  }
}
```

```
def f() = {
  // allocation!
  val range = new Range(0, 10)
  // allocation!
  val closure = (i) => println(i)
  range.foreach(closure)
}
```

Figure 7: A decompiled `for` loop in Scala: a simple `for` loop (left) decompiles into code that performs multiple heap allocations (right), potentially inhibiting function inlining. The decompiled code is rewritten in Scala for brevity.

In addition to Ghidra, we used JProfiler, a popular tool among Java developers for analyzing and visualizing runtime performance, to identify performance bottlenecks and track global heap allocations. For example, we observed that most of our computation time was spent in the calculation of $\Phi(n)$ (cf. line 8 of Figure 4), which enabled us to gain large performance improvements by focusing our optimization efforts on this calculation. Our overall observations from bytecode analysis and profiling led us to replace Breeze with our own custom row-major matrix class tailored for CP-APR operations. After our optimization efforts, we observed that the final single-threaded Scala implementation was roughly 9x faster than the MATLAB implementation.

3.3 JVM Restrictions and `sun.misc.Unsafe`

As a developer convenience, the JVM implements garbage collection, which frees the programmer from the responsibility of manually allocating and freeing memory. However, this flexibility comes at a cost: the memory layout of the heap must be managed by the garbage collector. This can be problematic for programs that require precise control over memory allocation. For example, within the CP-APR algorithm,

\(^1\)https://github.com/NationalSecurityAgency/ghidra
computing \( \Pi \) (cf. line 6 of Figure 4) requires a matrix with \( nnz \) rows and \( R \) columns, where \( nnz \) is the number of non-zeros in the tensor and \( R \) is the rank of the decomposition. As shown in Table 1, many of the tensors we decompose contain tens or hundreds of millions of non-zeros. With such large sparse tensors, the allocation of \( \Pi \) can be a point of failure for the JVM. In particular, we observed that the allocation of \( \Pi \) within our implementation would throw an OutOfMemoryException for any tensor larger than the Chicago tensor in Table 1, a relatively small tensor with roughly 5.2 million non-zeros. Since many of these tensors are larger than the Chicago tensor, it was necessary to find a workaround for this issue.

As an initial solution, we increased the maximum size of the JVM heap at program startup. However, we observed that this does not necessarily guarantee that requests for large allocations will be accommodated. For example, if the maximum heap size is set to 8 GB, an array allocation of 4 GB may still fail because the garbage collector may partition the available JVM heap memory in such a way that there is no 4 GB region of contiguous memory available for usage. After experimenting with successively larger maximum heap sizes to no avail, we concluded that it was a fruitless game of cat and mouse to negotiate with the JVM.

As a more permanent solution to circumventing the JVM’s limitations, we chose to use \texttt{sun.misc.Unsafe}, an internal class within the Java Development Kit (JDK) that exposes low-level implementation details about the target platform. In particular, \texttt{sun.misc.Unsafe} provides direct access to the C library’s \texttt{malloc} and \texttt{free} routines, allowing the developer to manage memory off the JVM heap via raw pointers. While the usage of this class is strongly discouraged by the JDK developers, in practice, many popular Java projects in the open-source community use it for performance reasons. For example, several of Apache Spark’s direct dependencies, including Apache Commons IO\(^2\) and Netty\(^3\), use \texttt{sun.misc.Unsafe} internally to clean up memory-mapped buffers and optimize memory accesses.

Despite \texttt{sun.misc.Unsafe} presenting an obvious solution to our problem, porting our implementation to use it was non-trivial and fraught with peril. For example, migrating away from JVM arrays to raw pointers created subtle bugs in address calculation because JVM arrays must be accessed with an index of type \texttt{int}, whereas elements behind a raw pointer must be accessed using pointer arithmetic on absolute addresses of type \texttt{long}. Since integer literals are of type \texttt{int} by default in Scala, this resulted in subtle overflow conditions that were not obvious from reading the source code. In addition to implementation bugs, we observed a 2x regression in performance using raw pointers. Through more profiling, we identified address calculation as the culprit; the calculations were occurring at the bytecode level rather than within the JVM’s native code. By optimizing our pointer arithmetic operations, we were able to win back all of our performance losses and then some, thanks to the unchecked nature of dereferencing raw pointers. Ultimately, while incorporating \texttt{sun.misc.Unsafe} into our code severely detracted from its readability and our overall developer productivity, we finally had the capability to decompose large tensors with CP-APR.

### 3.4 ALTO Implementation

Although ALTO improves sparse tensor decomposition performance, the format is not zero-cost. To perform any tensor computation, such as a Matrixized Tensor Times Khatri-Rao Product (MTTKRP) \cite{[1]}, a specific COO index for a given mode must first be extracted from its corresponding compressed ALTO form. This extraction is performed via a single \texttt{PEXT} instruction, which is found in the Bit Manipulation Instruction set (BMI2) for x86-64 processors. Although \texttt{PEXT} does create some overhead, it is minimal because \texttt{PEXT} is performed as a single assembly instruction with a hardware level implementation. It is expected that introducing the ALTO format to CP-APR would lead to large performance gains for sparse tensor decomposition with little overhead. However, the performance of ALTO is dependent on the ability to access BMI2 functionality.

Using Scala, ALTO’s implementation is restricted to the functionality provided by the JVM. Since the JVM does not provide access to BMI2, a software level implementation using loops and bit manipulation was required to port ALTO to Scala. This software implementation was likely to reduce the performance of ALTO due to an increase in the overhead required to perform index extraction. However, through the JVM, Scala uses a JIT for frequently executed code. The JIT compiles JVM bytecode to native machine code at runtime, where sections of code that are executed frequently receive further levels of optimizations. Using the JIT, we expected the compiler to recognize our software level implementation of \texttt{PEXT} and produce a single

\(^2\)https://github.com/apache/commons-io
\(^3\)https://github.com/netty/netty
assembly instruction. However, as seen in Figure 8, even with the highest optimization level enabled for the JIT, this was not the case. Using the HotSpot disassembler\(^4\) to produce assembly generated at runtime by the JIT, it was clear that manually implementing these functions in software would considerably increase ALTO overhead and reduce performance. Section 4 will further discuss the performance regression of using ALTO in Scala due to this limitation.

```assembly
0x00007fc951c26a80: sub  $t0,$t0
0x00007fc951c26a90: add  $t1,$t1
0x00007fc951c26aa0: mov  $t2,$t2
0x00007fc951c26ab0: mov  $t3,$t3
0x00007fc951c26ac0: mov  $t4,$t4
0x00007fc951c26ad0: mov  $t5,$t5
```

Figure 8: Compiled and optimized assembly code for the software implementation of PEXT generated by the HotSpot disassembler. This function can be completed with a single line of assembly. However, a PEXT function written in software results in over 26 lines of assembly.

Working within the JVM, the ALTO format is also restricted to a 64-bit compressed representation for all sparse tensors. Other languages such as C/C++ can utilize 128-bit SIMD registers for compressing larger tensors to increase the range of sparse tensors that can utilize the ALTO format. This limitation can be overcome, using either Scala’s BigList class or two 64-bit integer values to construct a software version of a 128-bit value. Although both methods would allow for larger tensors to utilize ALTO, they would further increase ALTO’s overhead in Scala. Additionally, larger tensors that require a 128-bit implementation of ALTO would exponentially exacerbate the performance overhead caused by a software implementation of PEXT. Due to both of these factors, we determined it was not worthwhile to implement a 128-bit variation of ALTO in Scala. Similarly, when parallelizing CP-APR with Apache Spark, the ALTO format was not utilized due to the PEXT overhead.

### 3.5 Spark Implementation and Manual Threading

To parallelize our single-threaded implementation of CP-APR with Spark, we adopted the following approach that parallelizes the calculation of \(\Phi^{(n)}\), the main bottleneck of the algorithm, for \(K\) workers. Note that \(\Pi^{(n)}\) denotes the matrix whose rows are given by those of \(\Pi^{(n)}\) associated with the non-zeros assigned to the \(k\)-th worker, and \(\Phi^{(n)}\) denotes the matrix that is the \(k\)-th summand of \(\Phi^{(n)}\), associated with the same non-zeros, such that that \(\sum_{k=1}^{K} \Phi^{(n)} = \Phi^{(n)}\).

1. Partition the non-zeros of the sparse tensor into \(K\) roughly equal sized partitions.
2. Compute the \(\Pi^{(n)}\) matrices in parallel.

\(^4\)https://github.com/jinzhengyang/hasis
(3) Compute the $\Phi^{(n)}_k$ matrices using the corresponding $\Pi^{(n)}_k$ matrices in parallel.

(4) Sum up each $\Phi^{(n)}_k$ to obtain $\Phi^{(n)}$.

Thanks to Spark's high-level API, making these changes to our single-threaded COO implementation was straightforward. However, we observed a nearly 100x regression with our initial implementation for a single worker. After consulting the Spark performance tuning documentation and our Spark application logs, we identified worker communication as the primary bottleneck. To address this issue, we leveraged Spark's broadcasting API to more efficiently distribute sparse tensor data to workers, and we swapped out the default serializer for Kryo\(^5\), an open-source Java library that uses `sun.misc.Unsafe` internally to speed up serialization. We also experimented with recomputing the rows of $\Pi^{(n)}_k$ on-the-fly instead of storing them to minimize worker communication, but this resulted in unacceptable regressions in performance. Through our efforts, we reduced the runtime to only be roughly 2x slower than the single-threaded implementation.

Unfortunately, using Spark still had several unavoidable pitfalls. For example, Spark requires mutable local variables to be allocated within the scope of the task, so each matrix computation requires a new allocation for each iteration. Furthermore, since Spark must serialize values returned from each transformation, it is impossible to guarantee that the raw pointers described in Section 3.3 can be safely freed. In conjunction with the allocation occurring each iteration, this can quickly lead to out-of-memory issues. As a result, we reverted back to using JVM arrays and only tested a handful of the data sets in Table 1 for our Spark benchmarks; a further discussion on performance and scalability will follow in Section 4.

In general, our parallelized CP-APR algorithm does not conform well to Spark's computational paradigm of relatively few reduction operations and fine-grained parallelism. For example, since the $\Phi^{(n)}_k$'s are added to produce $\Phi^{(n)}$ within the innermost loop of CP-APR, workers may be dispatched thousands of times. Furthermore, instead of letting Spark adaptively partition the sparse tensor, we were forced to manually specify each partition because each worker requires a precise upper bound on the number of non-zeros in its assigned partition to allocate enough space for the computation of $\Pi^{(n)}_k$. This is a stark contrast to how Spark is used in practice for exploratory data analysis, where users execute queries that operate on the rows of a SQL-like database representation of the data set, with a handful of reduction operations to aggregate results back on the main thread or node.

While we successfully implemented a parallel version of CP-APR with Spark, we were forced to leave a lot of performance improvements on the table, especially in the context of computation on a single node. To further investigate the performance overhead of Apache Spark, we decided to rewrite our parallel CP-APR algorithm with manual threading. To prototype the initial algorithm, we chose to utilize Rust, a systems programming language from Mozilla Research with a focus on safety and performance. Our motivation is twofold: Rust eliminates a large class of concurrency bugs at compilation time, and it gave us a baseline to benchmark our JVM-based implementation against native code emitted by LLVM. With our Rust rewrite in hand, it was trivial to port the changes to Scala. Both implementations demonstrate superior performance relative to the Spark implementation; a further discussion will follow in Section 4.

4 Performance Evaluation

In this section, we present a performance evaluation of our CP-APR implementations. Our goal is to demonstrate the efficacy of using high productivity frameworks for tensor decomposition.

4.1 Experimental Setup

All of our experiments were executed on a workstation with two Intel Xeon Haswell CPUs (total of 20 cores) and 512 GB of DDR4 memory. OpenJDK version 1.8.0.312 (Java 8) was used for compilation and JVM execution. All tensor decompositions were of rank 10, and the average execution time across three runs is recorded in all tables. Table 1 displays the properties of the various tensors used for testing. All tensors can be found in the Formidable Repository Of Tensors and Tools (FROSTT) \(|\) and were collected from real-world sources. Across the test suite, it is important to note that the number of non-zeros and densities vary widely from tensor to tensor.

\(^5\)https://github.com/EsotericSoftware/kryo
Table 1: List of tensors and properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Dimensions</th>
<th>Non-Zeros</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBNL</td>
<td>2K x 4K x 2K x 4K x 868K</td>
<td>1.6M</td>
<td>4.23e-14</td>
</tr>
<tr>
<td>NIPS</td>
<td>2K x 3K x 14K x 17</td>
<td>3.1M</td>
<td>1.82e-06</td>
</tr>
<tr>
<td>UBER</td>
<td>183 x 24 x 1K x 2K</td>
<td>3.3M</td>
<td>3.85e-4</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>6K x 24 x 77 x 32</td>
<td>5.3M</td>
<td>1.46e-2</td>
</tr>
<tr>
<td>VAST</td>
<td>165K x 11K x 2 x 100 x 89</td>
<td>26M</td>
<td>7.77e-07</td>
</tr>
<tr>
<td>ENRON</td>
<td>6K x 6K x 244K x 1K</td>
<td>51M</td>
<td>5.46e-09</td>
</tr>
<tr>
<td>NELL-2</td>
<td>12K x 9K x 29K</td>
<td>77M</td>
<td>2.40e-05</td>
</tr>
<tr>
<td>FLICKR</td>
<td>320K x 28M x 2M x 731</td>
<td>112M</td>
<td>1.07e-14</td>
</tr>
<tr>
<td>DELICIOUS</td>
<td>533K x 17M x 2M x 1K</td>
<td>140M</td>
<td>4.26e-15</td>
</tr>
<tr>
<td>NELL-1</td>
<td>3M x 2M x 25M</td>
<td>143M</td>
<td>9.05e-13</td>
</tr>
</tbody>
</table>

Figure 9: Speedup for Apache Spark implementation and COO format with up to 32 partitions

4.2 Results

In Figure 9, the term “partition” is used in accordance with the vocabulary found in the Apache Spark API to describe the number of parallel threads of execution. As the number of partitions increases, we observe positive scaling across all tensors, especially for the tensors that are more dense. Such dense tensors exhibit better cache locality and load balancing across the threads, leading to good scalability. Due to the limitations discussed in Section 3.5, only three tensors could be decomposed with our Apache Spark CP-APR implementation, without throwing an OutOfMemoryException. Therefore, despite the portability Apache Spark offers for HPC, the API provides limited usefulness for the decomposition of tensors formed from real-world data.

Figure 10 presents the scalability of our multi-threaded Scala CP-APR implementation with the COO format. For a majority of the tensors decomposed utilizing the COO format, we again see positive scaling across an increasing number of threads. Compared to our Spark implementation (cf. Figure 9), not only does the multi-threaded Scala implementation allow for the decomposition of all tensors in Table 1, but the rate of positive scaling also dramatically increases, with some decompositions exhibiting nearly a 14x speedup. Despite these improvements, four tensors in the testing suite are associated with minimal or even negative scaling. Knowing the shortcomings of COO for sparse tensors, and observing that these are the most sparse tensors in the test suite (cf. Table 1), this result is not entirely surprising.

Figure 11 displays the scalability of our multi-threaded Scala implementation that utilizes the ALTO format. As discussed in Section 3.4, ALTO compressed indices were limited to 64-bit integer representations. Consequently, we were not able to decompose some of the tensors in Table 1. Unfortunately, these tensors
are the most sparse tensors in the testing suite, and thus we were unable to fully analyze ALTO performance. However, utilizing the ALTO format with a multi-threaded implementation does lead to positive scaling for all tensors, resulting in nearly a 14x speedup for several tensors. Hence, ALTO enables greater scaling than COO, for many tensors.

<table>
<thead>
<tr>
<th>Type</th>
<th>Single Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>8 Threads</th>
<th>16 Threads</th>
<th>32 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark w/COO</td>
<td>1151.28(2(s)</td>
<td>833.1523(s)</td>
<td>525.1863(s)</td>
<td>334.898(s)</td>
<td>214.994(7(s)</td>
<td>169.0333(s)</td>
</tr>
<tr>
<td>ALTO</td>
<td>945.0863(s)</td>
<td>543.5387(s)</td>
<td>328.9107(s)</td>
<td>160.857(s)</td>
<td>90.65833(s)</td>
<td>81.093(s)</td>
</tr>
<tr>
<td>COO</td>
<td>652.045(s)</td>
<td>381.521(s)</td>
<td>233.0507(s)</td>
<td>117.8397(s)</td>
<td>70.64967(s)</td>
<td>53.90633(s)</td>
</tr>
<tr>
<td>Rust w/COO</td>
<td>597.4144(s)</td>
<td>328.5966(s)</td>
<td>202.215(s)</td>
<td>102.061(s)</td>
<td>59.1874(s)</td>
<td>43.8423(s)</td>
</tr>
</tbody>
</table>

Finally, Figure 12 and Table 2 present a comparison of raw execution times for the Chicago tensor across all implementations referenced in this paper. As previously noted, both Spark and ALTO show reduced performance due to the various limitations for each implementation discussed in Section 3. These runtime results also illustrate the differences between highly optimized Scala running on the JVM, and native machine code generated by the Rust compiler. As the number of threads increase, the difference in raw execution times appear to be very similar for several implementations. For instance, between the ALTO and COO formats in Scala, there appears to be marginal differences. However, Table 2 demonstrates that for 32 threads, Scala with the ALTO format is approximately 50% slower than Scala with the COO format, which is roughly 20% slower than the Rust implementation with the same format. The Chicago tensor used for comparison is relatively small compared to other data sets, leading to differences in execution time that could be perceived as inconsequential. Since many tensors are much larger, these differences will become much larger, reducing the effectiveness of several implementations.

5 Conclusions and Future Work

In this work, we implemented CP-APR using Apache Spark and evaluated its performance to assess Spark's viability as a platform for implementing portable tensor decompositions on HPC systems. Our initial Spark implementation was roughly an order of magnitude slower than Kolda’s MATLAB reference implementation, so we invested a significant amount of developer productivity into profiling and optimization. While our final Spark implementation demonstrated positive scaling across multiple cores with reasonable performance,
the limitations of the underlying JVM runtime still imposed non-trivial overhead and prevented us from benchmarking many tensors formed from real-world applications. By circumventing the JVM, we achieved performance comparable to optimized native code and could test arbitrarily large datasets, but this optimization proved to be incompatible with the Spark programming model. Furthermore, we found that using Spark on systems with low-latency interconnects (e.g., InfiniBand) is still an area of active research [7], so distributed computations may suffer from sub-optimal hardware utilization. In summary, we conclude that Spark is unsuitable for portable tensor decomposition implementations. For future work, we will investigate other HPC frameworks that are not subject to the restrictions of the JVM for implementing portable tensor decompositions.
References


