ALO-NMF: Accelerated Locality-Optimized Non-negative Matrix Factorization

Gordon E. Moon
Sandia National Laboratories
Albuquerque, NM, U.S.A
gemoon@sandia.gov

J. Austin Ellis
Sandia National Laboratories
Albuquerque, NM, U.S.A
johelli@sandia.gov

Aravind Sukumaran-Rajam
Washington State University
Pullman, WA, U.S.A
a.sukumaranrajam@wsu.edu

Srinivasan Parthasarathy
The Ohio State University
Columbus, OH, U.S.A
srini@cse.ohio-state.edu

P. Sadayappan
University of Utah
Salt Lake City, UT, U.S.A
saday@cs.utah.edu

ABSTRACT

Non-negative Matrix Factorization (NMF) is a key kernel for unsupervised dimension reduction used in a wide range of applications, including graph mining, recommender systems and natural language processing. Due to the compute-intensive nature of applications that must perform repeated NMF, several parallel implementations have been developed. However, existing parallel NMF algorithms have not addressed data locality optimizations, which are critical for high performance since data movement costs greatly exceed the cost of arithmetic/logic operations on current computer systems. In this paper, we present a novel optimization method for parallel NMF algorithm based on the HALS (Hierarchical Alternating Least Squares) scheme that incorporates algorithmic transformations to enhance data locality. Efficient realizations of the algorithm on multi-core CPUs and GPUs are developed, demonstrating a new Accelerated Locality-Optimized NMF (ALO-NMF) that obtains up to 2.29× lower data movement cost and up to 4.45× speedup over existing state-of-the-art parallel NMF algorithms.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**; **Non-negative matrix factorization**.

KEYWORDS

Parallel Non-negative Matrix Factorization; Dimensionality Reduction; Data Locality Optimization

ACM Reference Format:
Gordon E. Moon, J. Austin Ellis, Aravind Sukumaran-Rajam, Srinivasan Parthasarathy, and P. Sadayappan. 2020. ALO-NMF: Accelerated Locality-Optimized Non-negative Matrix Factorization. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394486.3403227

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403227

1 INTRODUCTION

Non-negative matrix factorization (NMF) has found renewed interest within the core database systems community on problems ranging from node embedding for graph mining [23, 28] to collaborative filtering for recommender systems [1, 11, 22] to topic modeling for text mining [25, 27]. Furthermore, it is often used as a key kernel or workload to evaluate new systems that blend machine learning with databases.

Given a non-negative matrix $A \in \mathbb{R}_+^{V \times D}$ and $K \ll \min(V, D)$, NMF finds two non-negative rank-K matrices $W \in \mathbb{R}_+^{V \times K}$ and $H \in \mathbb{R}_+^{K \times D}$, such that the product of W and H approximates A [16]:

$$A \approx WH \tag{1}$$

For example, when NMF is used for node embedding, given a node-to-node adjacency matrix A in which each non-zero element represents a connection between the nodes through the edge information, NMF generates dense node embeddings in low-dimensional factor matrices W and H. The effectiveness of the NMF node embeddings can be directly evaluated through applications such as node classification and link prediction tasks [10].

Several algorithms have been proposed for NMF. They all involve repeated, alternating updates of some elements of W interleaved with updates of some elements of H, with the constraint of non-negativity on the elements, until a suitable error norm (either Frobenius norm or Kullback-Leibler divergence) is lower than a desired threshold. Previously developed algorithms for NMF differ in the granularity of the number of elements of W that are updated before switching to updating some elements of H. Prior work compared the rates of convergence of alternate algorithms and the parallelization of those algorithms. However, to the best of our knowledge, the minimization of data movement through the memory hierarchy, using techniques like tiling, has not been previously addressed. With costs of data movement from memory being significantly higher than the cost of performing arithmetic operations on current processors, data locality is an important consideration.

In this paper, we address the issue of data locality optimization for NMF. An analysis of the computational components of the FAST-HALS (Hierarchical Alternating Least Squares) algorithm for NMF [3] is first performed to identify data movement overheads. The associativity of addition is used to judiciously reorder additive contributions in updating elements of W and H, to enable 3D tiling of a computationally intensive component of the algorithm. The analysis of data movement overheads as a function of tile size leads to a

model for selection of effective tile sizes. Parallel implementations of the new Accelerated Locality-Optimized NMF algorithm (called **ALO-NMF**) are presented for both GPUs and multi-core CPUs. An experimental evaluation with datasets used in prior studies demonstrates significant performance improvement over state-of-the-art alternatives available for parallel NMF.

The paper is organized as follows. In the next section, we present background on NMF and related prior work. In Section 3, we present a high-level overview of the ALO-NMF algorithm. Section 4 provides details of ALO-NMF for multi-core CPUs and GPUs. In Section 5, we compare the data movement cost for ALO-NMF and the original FAST-HALS algorithm. Thereafter, we discuss the approach to tile size selection based on data movement analysis. Section 6 compares the performance of ALO-NMF with existing state-of-the-art parallel implementations.

2 BACKGROUND AND RELATED WORK

2.1 Non-negative Matrix Factorization Algorithms

NMF seeks to solve the optimization problem of minimizing reconstruction error between a matrix A and its approximation WH. In order to measure the reconstruction error for NMF, Lee et al. [16] adopted various objective functions, such as the Frobenius norm given two matrices and Kullback-Leibler divergence given two probability distributions. The objective functions $D(A||WH)$ based on the Frobenius norm is defined in Equation 2.

$$D_F(A||WH) = \frac{1}{2}||A - WH||_F^2 = \frac{1}{2}\sum_{vd}(A_{vd} - (WH)_{vd})^2 \quad (2)$$

To efficiently minimize the objective functions (above), several variants of NMF algorithms have been developed: *Multiplicative Update (MU)*, *Additive Update (AU)*, *Hierarchical Alternating Least Squares (HALS)* and *Alternating Non-negative Least Squares with Block Principle Pivoting (ANLS-BPP)*. Table 1 describes the notations used in this paper.

Table 1: Common notations for NMF algorithms

Notation	Description
A	Non-negative matrix
W	Non-negative rank-K matrix factor
H	Non-negative rank-K matrix factor
V	Number of rows in A and W
D	Number of columns in A and H
K	Low rank
C	Cache size
T	Tile size

Multiplicative update (MU) and additive update (AU) proposed by Lee et al. [16] are the simplest NMF algorithms. The MU algorithm updates two rank-K non-negative matrices W and H based on multiplicative rules and ensures convergence. MU strictly conforms to non-negativity constraints on W and H because the elements of W and H that have zero value are not updated. Unlike the MU algorithm, the AU algorithm updates W and H based on the gradient descent method and avoids negative update values using a learning rate. However, some studies have reported that the MU and AU algorithms have weaknesses such as slow convergence and low convergence rate [9, 13, 18].

As an alternative to the MU and AU approaches, the Alternating Least Squares (ALS) algorithm uses the gradients of two objective functions with respect to W and H in order to update each of W and H, one after the other at each epoch. Cichocki et al. [4] proposed Hierarchical Alternating Least Squares (HALS), which hierarchically updates only one k-th row vector of $H \in \mathbb{R}_+^{K\times D}$ at a time and then uses it to update a corresponding k-th column vector of $W \in \mathbb{R}_+^{V\times K}$. In other words, HALS minimizes K pairs of local objective functions with respect to the K row vectors of H and K column vectors of W at each epoch. A standard HALS algorithm iteratively updates each row of H and each column of W in order within the innermost loop. Based on the standard HALS algorithm, Cichocki et al. [3] further proposed an extended version called the FAST-HALS algorithm as described in Algorithm 1. H_k and W_k denote the k-th row of H and the k-th column of W, respectively. FAST-HALS updates all rows of H before starting the update to all columns of W, instead of alternately updating each row of H and each column of W at a time. Compared to the MU algorithm, the FAST-HALS algorithm converges faster and produces a better solution, while maintaining a similar computational costs as reported in [8, 14].

Alternating Non-negative Least Squares (ANLS) is a special type of Alternating Least Squares (ALS) approach. Kim et al. [14] proposed an Alternating Non-negative Least Squares based Block Principal Pivoting (ANLS-BPP) algorithm. Under the Karush-Kuhn-Tucker (KKT) conditions, the ANLS-BPP algorithm iteratively finds the indices of non-zero elements (passive set) and zero elements (active set) in the optimal matrices until the KKT conditions are satisfied. The values of indices that correspond to the active set become zero, and the values of passive set are approximated by solving $min||A-WH||_F^2$, which is a standard Least Squares problem. Kim et al. [14] have shown that ANLS-BPP and FAST-HALS yield comparable convergence rates. Interestingly, FAST-HALS has also been found to converge faster than their ANLS-BPP implementation on real-word text datasets: 20 Newsgroups and TDT2 (refer to Figure 5.3 in Kim et al. [14]).

2.2 Related Work on Parallel NMF

Since most of the variations of NMF algorithm are highly compute-intensive, many previous efforts have sought to parallelize the NMF algorithms. Previous studies on parallelizing NMF can be broadly categorized into two groups based on implementation for multi-core CPUs [2, 5, 7, 12, 17, 19] versus GPUs [15, 20, 21]. Furthermore, each study used various NMF algorithms for parallel implementations.

Shared-Memory Multiprocessor. Battenberg et al. [2] introduced parallel NMF using the MU algorithm for an audio source separation task. Fairbanks et al. [7] adopted ANLS-BPP based NMF in order to find the structure of temporal behavior in a dynamic graph given vertex features. Both [2] and [7] developed the parallel NMF implementations on shared-memory multi-core CPUs using the Intel Math Kernel Library (MKL).

Distributed-Memory Systems. Dong et al. [5] demonstrated that MU algorithm with shared-memory based parallel implementation are limited by slow convergence. To overcome this problem, they devised an MPI implementation of MU based NMF that improves over Parallel NMF (PNMF) proposed by Robila et al. [24]. Different NMF algorithms have previously used tiling/blocking to minimize data

movement. Dong et al. [5] partitioned the two factor matrices, W and H, into smaller blocks and is distributed each block to different threads. Each block simultaneously updates corresponding sub-matrices of the W and H, and a reduction operation is performed by collective communication operations using MPI. Similarly, Liu et al. [19] proposed a matrix partition scheme that partitions the two factor matrices along the shorter dimension (K dimension) instead of the longer dimensions (V or D dimensions). Therefore, each matrix is divided into more partitions compared to partitioning along the longer dimension, so that data locality is increased and communication cost is decreased when performing the product of two matrices. Kannan et al. [12] minimized the communication cost by communicating only with the two factor matrices and other partitioned matrices among parallel threads. Based on the ANLS-BPP algorithm, their implementation also reduced the bandwidth and data latency using MPI collective communication operations. Given an input matrix A, they partitioned W and H into P multiple blocks (tiles) across the V and D dimensions, which are the number of rows in W and columns in H, respectively. In our tiling approach, W and H are partitioned across the K dimension, and the sizes of each block in W and H are $V\times(K/P)$ and $(K/P)\times D$, respectively. Therefore, the efficiency of our tiling strategy is associated with the K dimension in the two rank-K factor matrices. Our key contribution is not tiling/blocking itself, but converting matrix-vector operations to matrix-matrix operations. Tiling enables us to do the latter.

GPU Platform. Lopes et al. [20] propose several GPU-based parallel NMF implementations that use both MU and AU algorithms, for both Euclidean and KL divergence objective functions. Mejía-Roa et al. [21] present NMF-mGPU that performs MU based NMF on either a single GPU device or multiple GPU devices through MPI for a large-scale biological dataset. Koitka et al. [15] present MU and ALS based GPU implementations with binding for the R environment. To our knowledge, our paper is the first to develop a FAST-HALS based parallel NMF implementation for GPUs.

3 OVERVIEW OF APPROACH

In this section, we present a high-level overview of our approach to optimize NMF for data locality. We begin by describing the FAST-HALS algorithm [3], one of the fastest algorithms for NMF as demonstrated by previous comparison studies [14]. We analyze the data movement overheads from main memory, for different components of that algorithm, and identify the main bottlenecks. We then show how the algorithm can be adapted by exploiting the associativity of addition to make the computation effectively tileable to reduce data movement from memory, whereas the original form is not tileable.

3.1 Overview of FAST-HALS Algorithm

Algorithm 1 shows pseudo-code for the FAST-HALS algorithm [3] for NMF. It iteratively updates H and W, fully updating all entries in H (lines 3-6) and then updating all entries in W (lines 7-11) during each epoch, until convergence. While the updates to H and W are slightly different (due to normalization of W), each of the updates involves a pair of matrix-matrix products (lines 3/4 and 7/8 for H and W, respectively) and a sequential loop that steps through

Algorithm 1: FAST-HALS algorithm for NMF

Input: $A \in \mathbb{R}_+^{V\times D}$: non-negative matrix, ϵ: machine epsilon, E: total number of epochs

1 Initialize $W \in \mathbb{R}_+^{V\times K}$ and $H \in \mathbb{R}_+^{K\times D}$ with random non-negative numbers
2 **for** *epoch* = *1* **to** E **do**
▷ Updating H
3 $R \leftarrow A^T W$
4 $S \leftarrow W^T W$
5 **for** $k = 0$ **to** $K - 1$ **do**
6 $H_k \leftarrow \max(\epsilon, H_k + R_k - H^T S_k)$
▷ Updating W
7 $P \leftarrow AH^T$
8 $Q \leftarrow HH^T$
9 **for** $k = 0$ **to** $K - 1$ **do**
10 $W_k \leftarrow \max(\epsilon, W_k Q_{kk} + P_k - WQ_k)$
11 $W_k \leftarrow \frac{W_k}{||W_k||_2}$ ▷ Normalize W_k column vector with l_2 norm
12 **return** W, H

features (k loop) to update one row (column) of H (W) at a time. The computation within these k loops involves vector-vector operations and matrix-vector operations. From a computational complexity standpoint, the various matrix-matrix products and the sequential (K times) matrix-vector products all have cubic complexity ($O(N^3)$ if all matrices are square and of size $N \times N$). But as we show by analysis of data movement requirements in the next sub-section, the collection of matrix-vector products in lines 6 and 10 dominate. In the following sub-section, we present our approach to alleviate this bottleneck by exploiting the flexibility of instruction reordering via use of the associativity property of addition[1].

3.2 Data Movement Analysis for FAST-HALS Algorithm

The code regions with high data movement can be identified by individually analyzing each line in Algorithm 1. Lines 3 and 4 perform matrix multiplication. Given two matrices A^T, $(D \times V)$ and W, $(V \times K)$, it is well known that $2DKV/\sqrt{C}$ is the highest order term in the number of data elements moved (between main memory and a cache of size C words) for efficient tiled matrix-matrix multiplication[2]. Thus, the data movement costs associated with lines 3 and 4 are $2DKV/\sqrt{C}$ and $2KKV/\sqrt{C}$, respectively. The loop in line 5 performs matrix-vector multiplication and has an associated data movement cost of $K(3D+DK+K)$. Similar to lines 3 and 4, the data movement costs for lines 7 and 8 are $2VKD/\sqrt{C}$ and $2KKD/\sqrt{C}$, respectively. The loop in line 9 has an associated data movement cost of $K(VK + K + 6V + 1)$. The total data movement for Algorithm 1 is shown in Equation 3.

$$K\left(K(V+D)\left(1+\frac{2}{\sqrt{C}}\right)+\frac{4VD}{\sqrt{C}}+6V+3D+2K+1\right) \quad (3)$$

The main data movement overhead is associated with loops in lines 5 and 9. For example, the combined fractional data movement overhead of lines 6 (within loop in line 5) and 10 (within loop in line 9) is 91% for the 20 Newsgroups dataset. If the operational intensity (defined as the number of operations per data element moved) is very low, the performance will be bounded by memory bandwidth

[1] Floating-point addition is of course not strictly associative, but as shown later by the experimental results, the changed order does not adversely affect algorithm convergence.

[2] An extensive discussion of both lower bounds and data movement volume for several tiling schemes may be found in the work of Smith [26].

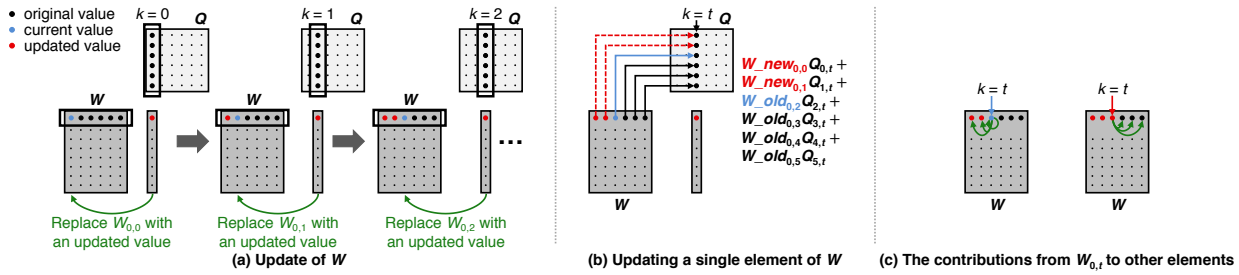

Figure 1: The interaction between different columns of W in the original FAST-HALS algorithm

and thus we will not be able to achieve the peak compute capacity. Due to its low operational intensity, the performance of Algorithm 1 is limited by the memory bandwidth. Hence, the major motivation for our algorithm adaptation is to achieve better performance by reducing the required data movement.

3.3 Overview of ALO-NMF

In this sub-section, we describe how the FAST-HALS algorithm is adapted by exploiting the flexibility of changing the order in which additive contributions to a data element are made. Before describing the adaptation, we first highlight the interaction between different columns of W with iterative matrix-vector operations in the original algorithm. Figure 1a depicts the update of W, which corresponds to lines 9 to 11 in Algorithm 1. The red dot in Figures 1a, 1b, and 1c shows a single updated element produced by the dot-product of a row vector of W and a column vector of Q. Each of green arrows in Figures 1a, 1c, and 2 indicate that a single output element/tile will be updated to the corresponding element/tile in W before performing the next dot-product.

In Algorithm 1, the t^{th} column of W is updated as the product of W with the t^{th} column of Q, i.e., a matrix-vector multiplication operation. Since the update to the $(t+1)^{th}$ column requires the modified W after the t^{th} column update, different columns (t: features) are updated sequentially. Let W_old represent the values at the beginning of the current epoch, and let W_new represent the values at the end of current epoch (updated values). The relationship between W_old and W_new is shown in Figure 1b, which depicts the contributions from W_old and W_new to $W\_new_{i,t}$. $W\_new_{i,t}$ can be obtained by $\sum_{j=0}^{t-1} W\_new_{i,j} \times Q_{j,t} + \sum_{j=t}^{K-1} W\_old_{i,j} \times Q_{j,t}$. Figure 1c shows the contributions of $W\_old_{i,t}$ and $W\_new_{i,t}$ to $W\_new_{i,*}$. $W\_old_{i,t}$ contributes to $W\_new_{i,j}$ $\forall j \mid j \leq t$, and $W\_new_{i,t}$ contributes to $W\_new_{i,j}$ where $\forall j \mid j > t$. In other words, the old value of column t is used to update the columns to the left of t (and self), and the new/updated value of column t is used to update the columns to the right of column t. If we partition W into a set of column panels (tiles) of size T, the interactions between columns can be expressed in terms of tiles as depicted in Figure 2. Similar to individual columns, the old value of data-tile τ is used to update the columns to the left of τ (phase 1), and the new/updated value of data-tile τ is used to update the data-tiles to the right of tile τ (phase 3). The updates to different columns within a data-tile (phase 2) are done sequentially.

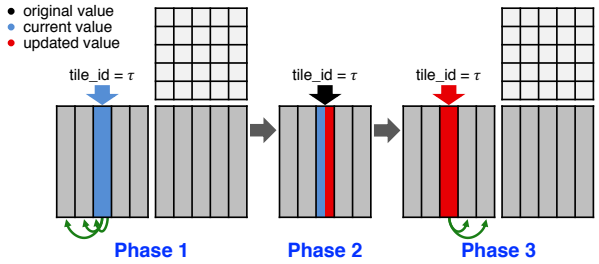

Figure 2: Overview of ALO-NMF for updating W.

The contributions to data-tiles to the left of current data-tile τ can be done as $W\_new_{i,j}$ -= $W\_old_{i,\tau\times T:((\tau+1)\times T)-1} \times Q_{\tau\times T:((\tau+1)\times T)-1,j}$ where $\forall j \mid j < \tau \times T$. Similarly, contributions to data-tiles to the right of current data-tile τ can be done as the following equation. $W\_new_{i,j}$ -= $W\_new_{i,\tau\times T:((\tau+1)\times T)-1} \times Q_{\tau\times T:((\tau+1)\times T)-1,j}$ where $\forall j \mid j \geq (\tau+1)\times T$. Both phases 1 and 3 use matrix-matrix operations which provide better performance and lower data movement than matrix-vector operations. Note that the total number of operations in both the original formulation and our formulation are exactly the same (refer to Table 3 in Section A).

4 DETAILS OF ALO-NMF ON MULTICORE CPUS AND GPUS

4.1 Parallel ALO-NMF CPU Implementation

Algorithm 2 shows ALO-NMF CPU pseudo-code for updating W. We begin by computing AH^T (line 1). Our algorithm does not distinguish between sparse matrix A and dense matrix A. The reason is that the sparsity only appears at sparse matrix-dense matrix multiplications (SpMM) in lines 3 and 7 in Algorithm 1, which always results in dense matrices R and P. Note that our key contribution is to optimize the main bottleneck involving iterative matrix-vector multiplications in lines 5 to 6 and lines 9 to 11 in Algorithm 1. Hence, our optimization remains the same for both sparse and dense matrix A. However, if A is sparse, then the actual implementation uses **mkl_dcsrmm**() and **cblas_dgemm**() otherwise. In the dense case, dense linear algebra libraries are used. It is possible to use dense libraries even when A is sparse (with zero filling), but is not beneficial for performance. Thereafter, line 2 in Algorithm 2 computes HH^T (using **cblas_dgemm**()). The values of W from the previous epoch are kept in W_old. We maintain another data structure called W_new which represents the updated values of W.

Algorithm 2: Parallel CPU implementation of updating W

Input: $A \in \mathbb{R}_{+}^{V \times D}$: input matrix, W_old and W_new: $V \times K$ non-negative matrix factor, H: $K \times D$ non-negative matrix factor, T: tile size, ϵ: machine epsilon

1 $P \leftarrow AH^T$
2 $Q \leftarrow HH^T$
▷ Initialize W_new using W_old and Q
3 **for** $i = 0$ **to** $V - 1$ **do**
4 **for** $j = 0$ **to** $K - 1$ **do**
5 $W\_new[i][j] \leftarrow W\_old[i][j] \times Q[j][j]$
6 $\gamma \leftarrow \text{ceil}(K/T)$ ▷ γ: total number of tiles
▷ **Phase 1**
7 **for** $tile\_id = 0$ **to** $\gamma - 1$ **do**
8 $W\_new[0 : V - 1][0 : tile\_id \times T - 1]$ -=
 dgemm($W\_old[0 : V - 1][tile\_id \times T : (tile\_id + 1) \times T - 1]$,
 $Q[tile\_id \times T : (tile\_id + 1) \times T - 1][0 : tile\_id \times T - 1]$)
▷ Phases 2 & 3
9 **for** $tile\_id = 0$ **to** $\gamma - 1$ **do**
▷ **Phase 2**
10 **for** $t = tile\_id \times T$ **to** $(tile\_id + 1) \times T - 1$ **do**
11 sum_square $\leftarrow 0$
12 **#pragma omp parallel for reduction(+:sum_square)**
13 **for** $i = 0$ **to** $V - 1$ **do**
14 sum $\leftarrow 0$
15 **#pragma omp simd reduction(+:sum)**
16 **for** $j = tile\_id \times T$ **to** $t - 1$ **do**
17 sum $\leftarrow$ sum + $W\_new[i][j] \times Q[t][j]$
18 **#pragma omp simd reduction(+:sum)**
19 **for** $j = t$ **to** $(tile\_id + 1) \times T - 1$ **do**
20 sum $\leftarrow$ sum + $W\_old[i][j] \times Q[t][j]$
21 $W\_new[i][t] \leftarrow \max(\epsilon, W\_new[i][t] + P[i][t] - \text{sum})$
22 sum_square $\leftarrow$ sum_square + $W\_new[i][t] \times W\_new[i][t]$
23 **#pragma omp parallel for**
24 **for** $i = 0$ **to** $V - 1$ **do**
25 $W\_new[i][t] \leftarrow W\_new[i][t]/\text{sqrt(sum\_square)}$
▷ **Phase 3**
26 $W\_new[0 : V - 1][(tile\_id + 1) \times T : K - 1]$ -=
 dgemm($W\_new[0 : V - 1][tile\_id \times T : (tile\_id + 1) \times T - 1]$,
 $Q[tile\_id \times T : (tile\_id + 1) \times T - 1][(tile\_id + 1) \times T : K - 1]$)

W_new is initialized by the loop in line 3.

$$W\_new[:, 0 : (\tau \times T) - 1] \text{ -=} \\ W\_old[:, (\tau \times T) : ((\tau + 1) \times T) - 1] \cdot \\ Q[(\tau \times T) : ((\tau + 1) \times T) - 1, 0 : (\tau \times T) - 1] \quad (4)$$

By using Equation 4, phase 1 is done by the loop in line 7. Figure 3 illustrates the computations of tiled matrix-matrix multiplications for three sequential phases, where τ denotes the index of the current tile and T is the size of each tile. For example, at current tile τ, phase 1 performs multiplication of the same colored/patterned two sub-matrices (tiles) in W_old and Q to update the result matrix W_new.

$$W\_new[:, (\tau \times T) : ((\tau + 1) \times T) - 1] \text{ -=} \\ W[:, (\tau \times T) : ((\tau + 1) \times T) - 1] \cdot \\ Q[(\tau \times T) : ((\tau + 1) \times T) - 1, (\tau \times T) : ((\tau + 1) \times T) - 1] \\ + P[:, (\tau \times T) : ((\tau + 1) \times T) - 1] \quad (5)$$

The loop in line 10 performs phase 2 computations as formulated in Equation 5. In order to take advantage of the vector units, the loops in lines 16 and 19 are vectorized. Additionally, a column-wise normalization for W_new is performed within phase 2 (lines 24 to 25).

$$W\_new[:, ((\tau + 1) \times T) : K - 1] \text{ -=} \\ W\_new[:, (\tau \times T) : ((\tau + 1) \times T) - 1] \cdot \\ Q[(\tau \times T) : ((\tau + 1) \times T) - 1, ((\tau + 1) \times T) : K - 1] \quad (6)$$

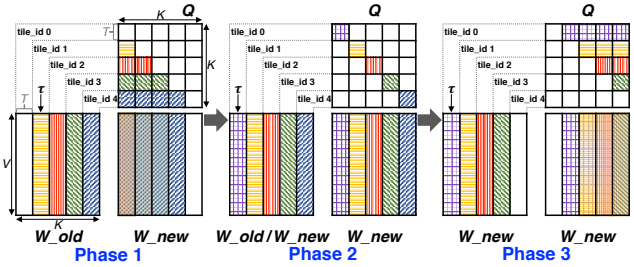

Figure 3: Computations of three phases for updating W.

The matrix-matrix multiplication in line 26 corresponds to the phase 3 computations using Equation 6. As depicted in Figure 3, the tiles involving phase 3 and phase 1 computations are different from each other. Finally, ALO-NMF CPU implementation completely substitutes lines 7 to 11 in Algorithm 1 for all lines in Algorithm 2. Similarly, H will be updated in the same fashion as updating W except for the normalization.

4.2 Parallel ALO-NMF GPU Implementation

Algorithm 3: GPU implementation of updating W on host

Input: $A \in \mathbb{R}_{+}^{V \times D}$: input matrix, W_old and W_new: $V \times K$ non-negative matrix factor, H: $K \times D$ non-negative matrix factor, T: tile size, ϵ: machine epsilon

1 $P \leftarrow AH^T$
2 $Q \leftarrow HH^T$
▷ Initialize W_new using W_old and Q
3 **init_W_new()** ▷ Refer to Algorithm 4 in Section A.1
4 $\gamma \leftarrow \text{ceil}(K/T)$ ▷ γ: total number of tiles
▷ **Phase 1**
5 **for** $tile\_id = 0$ **to** $\gamma - 1$ **do**
6 $W\_new[0 : V - 1][0 : tile\_id \times T - 1]$ -=
 cublasDgemm($W\_old[0 : V - 1][tile\_id \times T : (tile\_id + 1) \times T - 1]$,
 $Q[tile\_id \times T : (tile\_id + 1) \times T - 1][0 : tile\_id \times T - 1]$)
▷ Phases 2 & 3
7 **for** $tile\_id = 0$ **to** $\gamma - 1$ **do**
▷ **Phase 2**
8 **for** $t = tile\_id \times T$ **to** $(tile\_id + 1) \times T - 1$ **do**
9 cudaMemset(sum_square, 0)
10 **phase_2_W()** ▷ Refer to Algorithm 5 in Section A.1
11 __cudaDeviceSynchronize()
12 **norm_W_new()** ▷ Refer to Algorithm 6 in Section A.1
13 __cudaDeviceSynchronize()
▷ **Phase 3**
14 $W\_new[0 : V - 1][(tile\_id + 1) \times T : K - 1]$ -=
 cublasDgemm($W\_new[0 : V - 1][tile\_id \times T : (tile\_id + 1) \times T - 1]$,
 $Q[tile\_id \times T : (tile\_id + 1) \times T - 1][(tile\_id + 1) \times T : K - 1]$)

Similar to ALO-NMF CPU algorithm, ALO-NMF GPU algorithm also tries to minimize the data movement. Algorithm 3 and Algorithm 4, 5 and 6 in Section A.1 show the pseudo-code of ALO-NMF GPU algorithm. Since the overall structure of the GPU algorithm is similar to the CPU algorithm, this section only highlights the differences. Algorithm 3 runs on the host which is responsible for launching GPU kernels. The sparse matrix-dense matrix multiplication is implemented using **cusparseDcsrmm()**, and dense matrix-dense matrix multiplication is implemented using **cublasDgemm()**.

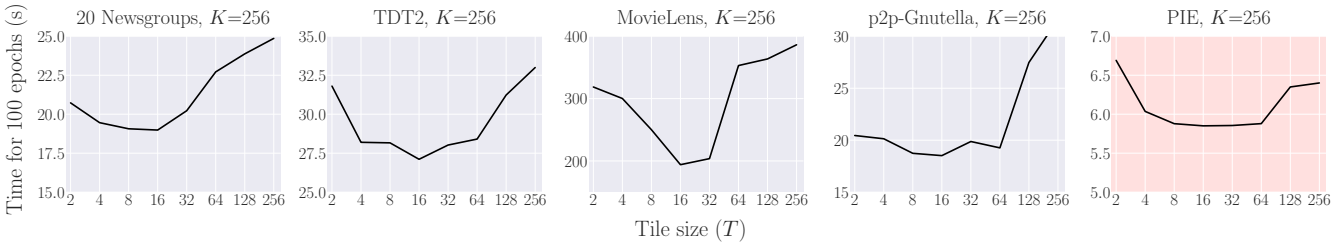

Figure 4: The training time in seconds for 100 epochs when the tile size T is varied for K=256 on five datasets. X-axis: tile size; Y-axis: elapsed time in seconds for 100 epochs. Each point is averaged over three executions.

Algorithm 5 in Section A.1 shows the pseudo-code for phase 2. In GPUs, the reduction across V (for normalization of W) can be performed using global memory atomic operations which are expensive. Hence, ALO-NMF GPU uses efficient hierarchical reduction. The reduction within a thread block is done in four steps: (i) in line 14 in Algorithm 5, the reduction across the threads within a warp is done using efficient warp shuffling primitives, (ii) all the threads with lane id 0 write the reduced value to shared memory (line 15), (iii) in line 18, the first warp of the thread block loads the previously written values from shared memory and (iv) all the threads in the first warp again performs warp reduction (line 20). In order to perform reduction across multiple thread blocks, we use atomic operations which is shown in line 22. Algorithm 6 in Section A.1 shows the pseudo-code for normalization.

5 MODELING: DETERMINATION OF THE TILE SIZE

In this section, we first compare the data movement cost of ALO-NMF with the original FAST-HALS algorithm. Then the data movement of ALO-NMF as a function of T is optimized to select effective tile sizes.

$$\sum_{i=0}^{\frac{K}{T}-1} iVT^2\Big(\frac{1}{T}+\frac{2}{\sqrt{C}}\Big) = VT^2\Big(\frac{1}{T}+\frac{2}{\sqrt{C}}\Big)\Big(\frac{K^2-KT}{2T^2}\Big) \tag{7}$$

$$\sum_{i=0}^{\frac{K}{T}-1} T(VT+T+V) = \frac{K}{T}T(VT+T+V) \tag{8}$$

In ALO-NMF, W is updated in three phases. Phases 1 and 3 can be implemented using matrix-multiplication, and the corresponding cost is shown in Equation 7, where T represents the tile size and C is the cache size. Phase 2 can be implemented using matrix-vector multiplication and the associated cost is shown in Equation 8. Since loading matrix W dominates the data movement cost in phase 2, the cost of loading vectors can be ignored.

$$vol(T) = V\Big(\frac{1}{T}+\frac{2}{\sqrt{C}}\Big)(K^2-KT)+\frac{K}{T}T(VT) \tag{9}$$

Equation 9 shows the total data movement required for updating W in ALO-NMF . The cost of updating H is similar to updating W, but updating H does not require accessing Q. In addition, since H is not normalized, the cost associated with normalization is not present. The data movement cost of the original loop in line 9 in Algorithm 1 is $K(VK+K+6V+1)$. Hence, for the dense PIE dataset (V=11,554) with low rank K=256 on a machine with 33 MB cache, the data movement cost of original scheme is 775,015,680 bytes. However, in our scheme based on Equation 9, the cost is only 338,840,256 bytes which is 2.29× lower than the original scheme (when T=16 is selected for K=256).

The tile size T affects the data movement volume and hence the performance. Given the data movement of our algorithm as a function of T (Equation 9), consider the case when there is only one tile (T=K). In this case, there is no work associated with phase 1 (contributions to left) and phase 3 (contributions to the right) as the first term of Equation 9 will become zero. The total data movement of phase 2 is VK^2 which is very high. Thus, when T is high, the total data movements required for phases 1 and 3 are low, but phase 2 has high data movement. Now consider the other extreme where the tile size is 1 (T=1). When T is low, the total data movements for phases 1 and 3 are high, but phase 2 has low data movement. Hence, we expect the combined data movement for all the phases to decrease as T increases from 1 to some point and then the data movement will increase again as T approaches K. Figure 4 shows the performance results across different tile sizes for K=256 on five datasets. Since performance is correlated with data movement, the performance as a function of tile size T should show a similar trend with the performance shown in Figure 4.

$$\frac{d(vol(T))}{dT} = T^2\Big(\frac{2}{\sqrt{C}}-1\Big)+K = 0 \tag{10}$$

$$T = \sqrt{\frac{K\sqrt{C}}{\sqrt{C}-2}} \tag{11}$$

In order to build a model to determine the best tile size for a given K, the derivative of Equation 9 with respect to T is set it to zero as shown in Equation 10. The solution to Equation 10 is shown in Equation 11. As a result, for a machine with cache size of 33 MB, the predicted tile size computed by our model (Equation 11) is 19.82 for K=256. As shown in Figure 4, tile size selected by our model is optimal/near optimal. For example, when K=256, evaluation shows that the best performance is achieved for T=16, which is very close to our model predicted tile size of T=19.82.

6 EXPERIMENTAL EVALUATION

This section compares the time to convergence and convergence rate of ALO-NMF with various state-of-the-art NMF algorithms. All the CPU experiments were run on a 24-core Intel(R) Xeon(R) Platinum 8160 CPU running at 2.10 GHz. The GPU experiments were run on an NVIDIA Tesla P100 SXM2 GPU with 16 GB global

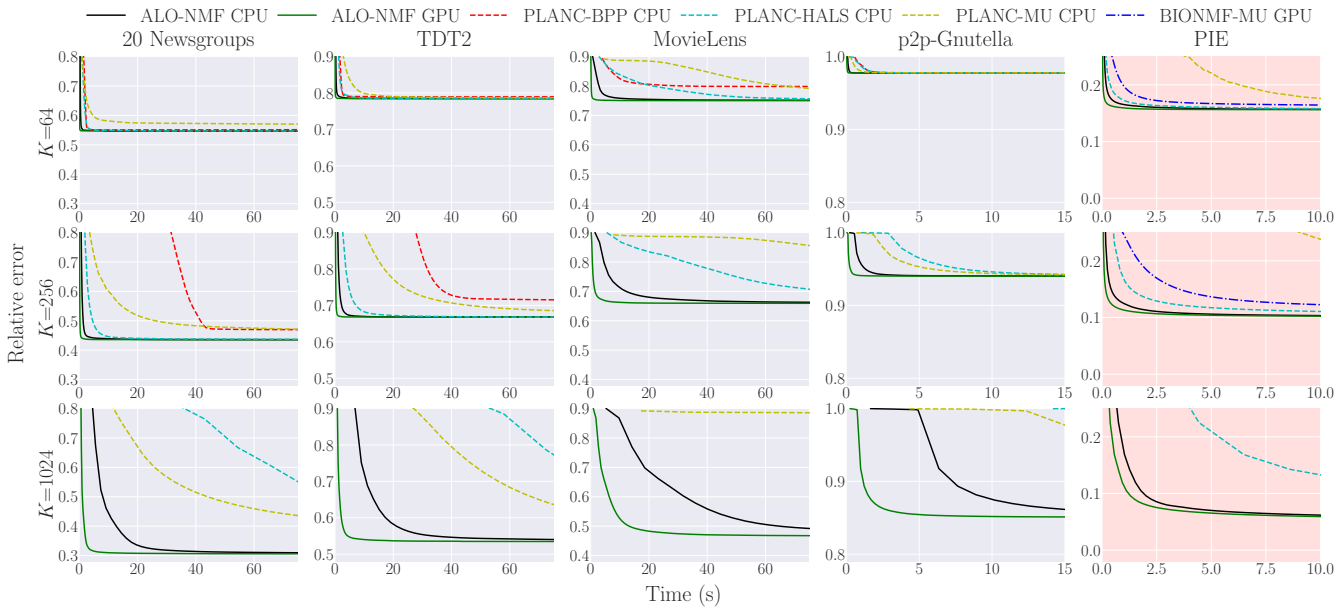

Figure 5: Relative objective value over training time on five datasets. Grey and red backgrounds indicate sparse and dense datasets, respectively. According to current model, the T values for K=64, 256 and 1024 are set to 8, 16 and 64, respectively. X-axis: elapsed time in seconds; Y-axis: relative error.

memory. Table 4 in Section A.2 details the benchmarking machine configurations in the experiments.

For experimental evaluations we used two publicly available real-world text datasets – 20 Newsgroups[3] and TDT2[3]. In addition, we used a real-world directed graph dataset – p2p-Gnutella[4] and a rating dataset – MovieLens[5]. In order to represent the audio-visual context analysis in social media platforms, we used an image dataset – PIE[3]. 20 Newsgroups, TDT2, MovieLens and p2p-Gnutella are sparse matrices, and PIE is a dense matrix (more details on the datasets are provided in Section A.3). In order to evaluate the accuracy of different NMF models, we measured the relative objective function $\sqrt{\sum_{vd}(A_{vd} - (WH)_{vd})^2 / \sum_{vd}(A_{vd})^2}$ suggested by Kim et al. [14], where A_{vd} and $(WH)_{vd}$ denote the values of each element in an input matrix $A \in \mathbb{R}_+^{V \times D}$ and an approximated matrix $(WH) \in \mathbb{R}_+^{V \times D}$, respectively. The capability of each NMF model in minimizing the objective function can be obtained by measuring relative changes of objective value over epochs.

6.1 NMF Implementations Compared

We compared ALO-NMF on CPUs and GPUs with the state-of-the-art parallel NMF implementations such as PLANC[6] by Kannan et al. [7, 12] and BIONMF[7] by Mejía-Roa et al. [21]. The four implementations used in our comparisons are as follows:

- **PLANC-BPP CPU**: PLANC's OpenMP-based ANLS-BPP
- **PLANC-HALS CPU**: PLANC's OpenMP-based HALS
- **PLANC-MU CPU**: PLANC's OpenMP-based MU
- **BIONMF-MU GPU**: BIONMF's GPU-based MU

All CPU implementations, including PLANC-BPP CPU, PLANC-HALS CPU and PLANC-MU CPU, and our ALO-NMF CPU, used Intel's Math Kernel Library (MKL) for all BLAS (Basic Linear Algebra Subprograms) operations. Similarly, all GPU implementations, including BIONMF-MU GPU and our ALO-NMF GPU, used NVIDIA's cuBLAS library.

6.2 Performance Evaluation

Convergence. Figure 5 shows the relative error as a function of elapsed time for various NMF implementations for different K values. We used 48 (24 cores × 2 threads per core) threads for all CPU experiments. The tile size T was varied for each K, where $T \leq K$. For each dataset, the same randomly initialized non-negative matrices were used to evaluate all CPU and GPU implementations. Since the BIONMF-MU GPU implementation does not allow the input matrix to be sparse, we only compared ALO-NMF GPU with BIONMF-MU GPU on the dense PIE image dataset. In addition, BIONMF-MU GPU failed to execute when $K > 512$. ALO-NMF CPU and ALO-NMF GPU consistently outperformed existing state-of-the-art CPU and GPU implementations on all datasets. As reported in previous studies, FAST-HALS produced a better convergence rate than other NMF variants. MU and ANLS-BPP algorithms suffered from a lower convergence rate on both sparse and dense matrices. As shown in Figure 6, PLANC-HALS CPU was the only implementation which was able to maintain the same solution quality as ALO-NMF. However, ALO-NMF converged faster as shown in Figure 5. Although the same initialization was used for all NMF variants, convergence rates vary even among different implementations of the same NMF

[3] http://www.cad.zju.edu.cn/home/dengcai/Data/data.html
[4] http://snap.stanford.edu/data/p2p-Gnutella30.html
[5] https://grouplens.org/datasets/movielens/
[6] https://github.com/ramkikannan/planc
[7] https://github.com/bioinfo-cnb/bionmf-gpu

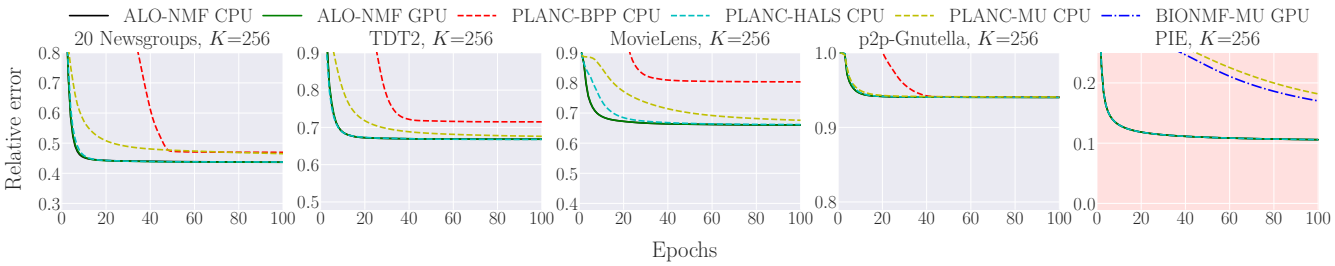

Figure 6: Comparison of convergence over epochs on five datasets, K=256 and T=16. X-axis: number of epochs; Y-axis: relative error.

algorithm. The differences in Figure 6 are due to differences in the processing order of elements by the different CPU/GPU parallelization schemes.

Speedup. Compared to the PLANC-HALS CPU, ALO-NMF CPU achieved 3.17×, 3.26×, 3.48×, 5.59×, and 4.45× speedup per epoch on the 20 Newsgroups, TDT2, MovieLens, p2p-Gnutella, and PIE datasets with K=256, respectively. As the relative error reduction per epoch is vastly different between MU and FAST-HALS algorithms, measuring the speedup per epoch between BIONMF-MU GPU and ALO-NMF GPU is not a fair comparison. Hence, the speedup of ALO-NMF GPU over all NMF variants to reach the same relative error is shown in Figure 7.

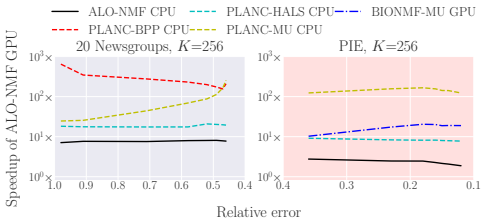

Figure 7: Speedup of ALO-NMF GPU over all CPU and GPU implementations on two datasets, K=256 and T=16.

Since the MU and ANLS-BPP algorithms perform different updates, the speedup of ALO-NMF GPU to obtain the same relative error value differs for different error values. However, when we only compare ALO-NMF GPU with HALS-based CPU implementations (ALO-NMF CPU and PLANC-HALS CPU), our ALO-NMF GPU maintains the same speedup over all the relative error values. In addition, all of the points in Figure 7 are greater than one, indicating that ALO-NMF GPU is the fastest of all competing implementations. For example, when the compared models, i.e., ALO-NMF CPU, PLANC-HALS CPU, BIONMF-MU GPU and PLANC-MU CPU, converge to the same 0.12 relative error, ALO-NMF GPU achieves 1.88×, 7.75×, 18.91× and 123.16× speedup on the PIE dataset, respectively.

Table 2 shows the elapsed time breakdown for each step in updating W. Both the original FAST-HALS based NMF[8] and ALO-NMF CPU use the same sparse and dense libraries for SpMM and

[8]We selected PLANC-HALS CPU as the baseline FAST-HALS NMF since it implements the FAST-HALS algorithm without blocking/tiling optimizations.

Table 2: Breakdown of elapsed time in seconds for updating a matrix W on the 20 Newsgroups dataset, K=256 and T=16. SpMM is the AH^T in line 7 in Algorithm 1 and line 1 in Algorithm 2. Similarly, DMM corresponds to the HH^T in line 8 in Algorithm 1 and line 2 in Algorithm 2.

	elapsed time (s)
SpMM	4.6e-2
DMM	1.2e-3
Iterative DMV	2.1e-1

(a) FAST-HALS based NMF

	elapsed time (s)
SpMM	4.6e-2
DMM	1.2e-3
Phase 1	2.0e-3
Phases 2 & 3	3.1e-2

(b) ALO-NMF CPU

DMM operations, respectively. The difference in updating W is that ALO-NMF CPU performs phases 1, 2 and 3 instead of iteratively computing DMV (dense matrix-vector multiplications). In the original FAST-HALS, the execution time is dominated by iterative DMV operations as shown in Table 2a. As expected, the total update time of W is significantly reduced by 68.76% in the ALO-NMF CPU algorithm. After applying our optimization, SpMM becomes the bottleneck. This indicates that the reformulation of the core-computations to matrix-matrix multiplication has yielded significant benefit.

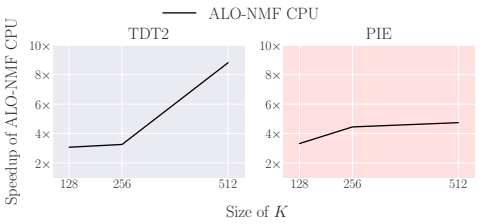

Figure 8: Speedup of ALO-NMF CPU over PLANC-HALS CPU when the sizes of K are varied on two datasets.

Figure 8 shows the scalability of ALO-NMF CPU against PLANC-HALS CPU for different values of K. Based on Equation 11, we used the tile sizes T={16, 16, 32} for K={128, 256, 512}, respectively. As K increases, our speedup also increases. For the TDT2 dataset, ALO-NMF CPU achieved approximately 3.07×, 3.26×, and 8.81× speedup over PLANC-HALS CPU when K=128, 256, and 512, respectively.

Achieved Peak Performance. Table 3 in Section A shows the total number of floating point operations required for the original FAST-HALS[8] and ALO-NMF algorithms. Given any tile size T, both equations in Table 3 produce exactly the same number of operations because ALO-NMF does not affect the required number of operations in HALS-based NMF. However, in terms of the achieved peak flops, ALO-NMF provides significant improvement. For the 20 Newsgroups dataset (V=26,214 and D=11,314) with K=256 and T=16, based on the equations in Table 3, the achieved peak flops of the original FAST-HALS is *total floating points/elapsed time(s)*$\times 10^{-9}$ =3.1e11/0.66(s)$\times 10^{-9}$=480 GFLOPs, whereas that of ALO-NMF CPU is 3.1e11/0.24(s)$\times 10^{-9}$=1300 GFLOPs which is approximately 2.7× higher than the original algorithm.

7 CONCLUSION

In this paper, we developed a HALS-based parallel ALO-NMF algorithm for multi-core CPUs and GPUs. The data movement overhead is a critical factor that affects performance. This paper does a systematic analysis of data movement overheads associated with NMF algorithm to determine the bottlenecks. ALO-NMF alleviates the data movement overheads by enhancing data locality. ALO-NMF achieved 2.29× lower data movement cost compared to the original FAST-HALS algorithm. Since efficiency of our tiling strategy is correlated to the K dimension in the factor matrices W and H, ALO-NMF provides significant performance improvement as the K size increases. Furthermore, our optimization technique can be applied to many other scientific computations not limited to machine learning kernels. Experimental results demonstrate ALO-NMF converged 4.45× faster than existing state-of-the-art parallel implementations while maintaining evaluation quality. We plan to extend this work by using these ideas in a semi-supervised setting [22], adding a performance portable distributed implementation [6], and handling massive database cases.

ACKNOWLEDGMENTS

We thank the ASC Advanced Architectures test-bed team at Sandia National Laboratories (SNL) for supplying and supporting the systems used in this paper. This work was supported in part by the U.S. National Science Foundation (NSF) through awards EAR-1520870, CCF-2018016, and SES-1949037. The findings expressed in this article are those of the author(s) and do not necessarily reflect the views of the NSF or SNL.

REFERENCES

[1] Mehdi Hosseinzadeh Aghdam, Morteza Analoui, and Peyman Kabiri. 2015. A Novel Non-negative Matrix Factorization Method for Recommender Systems. *Applied Mathematics & Information Sciences* 9, 5 (2015), 2721.

[2] Eric Battenberg and David Wessel. 2009. Accelerating Non-Negative Matrix Factorization for Audio Source Separation on Multi-Core and Many-Core Architectures.. In *ISMIR*. 501–506.

[3] Andrzej Cichocki and Anh-Huy Phan. 2009. Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 92, 3 (2009), 708–721.

[4] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. 2007. Hierarchical ALS Algorithms for Nonnegative Matrix and 3D Tensor Factorization. In *International Conference on Independent Component Analysis and Signal Separation*. Springer, 169–176.

[5] Chao Dong, Huijie Zhao, and Wei Wang. 2010. Parallel Nonnegative Matrix Factorization Algorithm on the Distributed Memory Platform. *International Journal of Parallel Programming* 38, 2 (2010), 117–137.

[6] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[7] James P Fairbanks, Ramakrishnan Kannan, Haesun Park, and David A Bader. 2015. Behavioral Clusters in Dynamic Graphs. *Parallel Comput.* 47 (2015), 38–50.

[8] Nicolas Gillis. 2014. The Why and How of Nonnegative Matrix Factorization. *Regularization, Optimization, Kernels, and Support Vector Machines* 12, 257 (2014).

[9] Edward F Gonzalez and Yin Zhang. 2005. *Accelerating the Lee-Seung Algorithm for Nonnegative Matrix Factorization.* Technical Report.

[10] Saket Gurukar, Priyesh Vijayan, Aakash Srinivasan, Goonmeet Bajaj, Chen Cai, Moniba Keymanesh, Saravana Kumar, Pranav Maneriker, Anasua Mitra, Vedang Patel, Balaraman Ravindran, and Srinivasan Parthasarathy. 2019. Network Representation Learning: Consolidation and renewed bearing. *arXiv preprint arXiv:1905.00987* (2019).

[11] Antonio Hernando, Jesús Bobadilla, and Fernando Ortega. 2016. A Non Negative Matrix Factorization for Collaborative Filtering Recommender Systems based on a Bayesian Probabilistic Model. *Knowledge-Based Systems* 97 (2016), 188–202.

[12] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization. *ACM SIGPLAN Notices* 51, 8 (2016), 1–11.

[13] Hyunsoo Kim and Haesun Park. 2008. Nonnegative Matrix Factorization based on Alternating Nonnegativity Constrained Least Squares and Active Set Method. *SIAM J. Matrix Anal. Appl.* 30, 2 (2008), 713–730.

[14] Jingu Kim and Haesun Park. 2011. Fast Nonnegative Matrix Factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing* 33, 6 (2011), 3261–3281.

[15] Sven Koitka and Christoph M Friedrich. 2016. nmfgpu4R: GPU-accelerated computation of the non-negative matrix factorization (NMF) using CUDA capable hardware. *The R Journal* 8, 2 (2016), 382–392.

[16] Daniel D Lee and H Sebastian Seung. 2001. Algorithms for Non-negative Matrix Factorization. In *Advances in Neural Information Processing Systems*. 556–562.

[17] Ruiqi Liao, Yifan Zhang, Jihong Guan, and Shuigeng Zhou. 2014. CloudNMF: A MapReduce implementation of nonnegative matrix factorization for large-scale biological datasets. *Genomics, Proteomics & Bioinformatics* 12, 1 (2014), 48–51.

[18] Chih-Jen Lin. 2007. Projected Gradient Methods for Nonnegative Matrix Factorization. *Neural Computation* 19, 10 (2007), 2756–2779.

[19] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. 2010. Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, 681–690.

[20] Noel Lopes and Bernardete Ribeiro. 2010. Non-negative Matrix Factorization Implementation using Graphic Processing Units. In *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 275–283.

[21] Edgardo Mejía-Roa, Daniel Tabas-Madrid, Javier Setoain, Carlos García, Francisco Tirado, and Alberto Pascual-Montano. 2015. NMF-mGPU: Non-negative matrix factorization on multi-GPU systems. *BMC Bioinformatics* 16, 1 (2015), 43.

[22] Anasua Mitra, Priyesh Vijayan, Srinivasan Parthasarathy, and Balaraman Ravindran. 2020. A Unified Non-Negative Matrix Factorization Framework for Semi Supervised Learning on Graphs. In *Proceedings of the 2020 SIAM International Conference on Data Mining, SDM 2020*, Carlotta Demeniconi and Nitesh V. Chawla (Eds.). 487–495.

[23] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying deepwalk, line, pte, and node2vec. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 459–467.

[24] Stefan A Robila and Lukasz G Maciak. 2006. A Parallel Unmixing Algorithm for Hyperspectral Images. In *Intelligent Robots and Computer Vision XXIV: Algorithms, Techniques, and Active Vision*, Vol. 6384. International Society for Optics and Photonics, 63840F.

[25] Tian Shi, Kyeongpil Kang, Jaegul Choo, and Chandan K Reddy. 2018. Short-Text Topic Modeling via Non-negative Matrix Factorization Enriched with Local Word-Context Correlations. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1105–1114.

[26] Tyler Michael Smith et al. 2018. *Theory and Practice of Classical Matrix-Matrix Multiplication for Hierarchical Memory Architectures.* Ph.D. Dissertation.

[27] Sangho Suh, Jaegul Choo, Joonseok Lee, and Chandan K Reddy. 2017. Local Topic Discovery via Boosted Ensemble of Nonnegative Matrix Factorization. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 4944–4948.

[28] Xiao Wang, Peng Cui, Jing Wang, Jian Pei, Wenwu Zhu, and Shiqiang Yang. 2017. Community Preserving Network Embedding. In *Thirty-first AAAI Conference on Artificial Intelligence*.

A APPENDIX

Table 3 lists the total number of floating point operations used to measure the achieved peak flops of the original FAST-HALS and ALO-NMF CPU implementations in Section 6.2.

Table 3: The total number of floating point operations

	Floating point operations
FAST-HALS	$2DVK + 2KVK + 2KDK + 3KD + 2VDK + 2KDK + 2KVK + 5KV + 2KV$
ALO-NMF	$2DVK + 2KVK + \sum_{i=0}^{\frac{K}{T}-1}(2DTiT) + \sum_{i=1}^{\frac{K}{T}}\left(2DTT + 3DT + 2DT\left(\frac{K}{T} - i\right)T\right) +$ $2VDK + 2KDK + \sum_{i=0}^{\frac{K}{T}-1}(2VTiT) + \sum_{i=1}^{\frac{K}{T}}\left(2VTT + 5VT + 2VT + 2VT\left(\frac{K}{T} - i\right)T\right)$

A.1 Pseudo-codes for ALO-NMF GPU implementation

Algorithm 4, 5 and 6 show the pseudo-code for the initialization, phase 2 operation and normalization in ALO-NMF GPU implementation, respectively.

Algorithm 4: init_W_new() kernel on GPUs

Input: W_old, W_new, Q, V, K
$vId \leftarrow$ blockIdx.x × blockDim.x + threadIdx.x ▷ thread ID
if $vId < V$ **then**
  for $j = 0$ **to** $K - 1$ **do**
    $W\_new[vId + j \times V] \leftarrow W\_old[vId + j \times V] \times Q[j + j \times K]$

Algorithm 5: phase_2_W() kernel on GPUs

Input: W_old, W_new, P, Q, sum_square, t, tile_id, T, V, K, ϵ
$vId \leftarrow$ blockIdx.x × blockDim.x + threadIdx.x ▷ thread ID
__shared__ $SHARED\_SUM$[1024/32]
sum_reduce ← 0.0f
if $vId < V$ **then**
  sum ← 0
  for $j = tile\_id \times T$ **to** $(tile\_id + 1) \times T - 1$ **do**
    if $j < t$ **then**
      sum ← sum + $W\_new[vId + j \times V] \times Q[t + j \times K]$
    else
      sum ← sum + $W\_old[vId + j \times V] \times Q[t + j \times K]$
  $W\_new[vId + t \times V] \leftarrow \max(\epsilon, W\_new[vId + t \times V] + P[vId + t \times V] - \text{sum})$
  sum_reduce ← $W\_new[vId + t \times V]$
sum_reduce ← sum_reduce × sum_reduce ▷ Warp-level reduction
sum_reduce ← **warp_reduce**(sum_reduce) ▷ Block-level reduction
if $threadIdx.x \% 32 == 0$ **then**
  $SHARED\_SUM$[threadIdx.x / 32] ← sum_reduce
__syncthreads()
if $threadIdx.x / 32 == 0$ **then**
  sum_reduce ← $SHARED\_SUM$[threadIdx.x]
  sum_reduce ← **warp_reduce**(sum_reduce)
if $threadIdx.x == 0$ **then**
  atomicAdd (sum_square, sum_reduce)

Algorithm 6: norm_W_new() kernel on GPUs

Input: W_new, sum_square, t, V
$vId \leftarrow$ blockIdx.x × blockDim.x + threadIdx.x ▷ thread ID
if $vId < V$ **then**
  $W\_new[vId + t \times V] \leftarrow W\_new[vId + t \times V]$/sqrt(sum_square)

A.2 Benchmarking Machines

Table 4 shows the configuration of the benchmarking machines used for experiments.

Table 4: Machine configuration

Machine	Details
CPU	Intel(R) Xeon(R) Platinum 8160 CPU (24 CPU cores, 2 threads per core) ICC version 19.5.281
GPU	Tesla P100 SXM2 (16 GB Global Memory, 56 SMs, 4 MB L2 cache) CUDA version 9.2.88

A.3 Datasets

The 20 Newsgroups dataset contains a document-term matrix in bag-of-words representation associated with 20 topics. TDT2 (Topic Detection and Tracking 2) dataset is a collection of text documents from CNN, ABC, NYT, APW, VOA and PRI. MovieLens dataset contains 10,000,054 movie ratings from the web-based movie recommender service called MovieLens. p2p-Gnutella is a sequence of snapshots from the Gnutella peer-to-peer file sharing network. Each node represents a Gnutella host. The directed graph encodes connections between the hosts. PIE dataset contains images of faces in dense matrix format. The size of each image in PIE dataset is 64×64 pixels. Table 5 shows the characteristics of each dataset.

Table 5: Statistics of datasets used in the experiments. V is the number of rows and D is the number of columns in non-negative matrix A. For the text datasets, V is the vocabulary size and D is the number of documents.

Dataset	V	D	Total NNZ
20 Newsgroups	26,214	11,314	1,018,191
TDT2	36,771	10,212	1,323,869
MovieLens	71,567	10,677	10,000,054
p2p-Gnutella	36,682	36,682	88,328
PIE	11,554	4,096	47,321,408