Distributed out-of-memory NMF on CPU/GPU architectures

Djibrilla Boureima, Ismael
Bhattarai, Manish
Eren, Maksim Ekin
Skau, Erik West
Romero, Phillip R.
Eidenbenz, Stephan Johannes
Alexandrov, Boian

Distributed out-of-memory NMF on CPU/GPU architectures

Ismael Boureima[1] · Manish Bhattarai[1] · Maksim Eren[1] · Erik Skau[2] · Philip Romero[3] · Stephan Eidenbenz[2] · Boian Alexandrov[1]

Abstract
We propose an efficient distributed out-of-memory implementation of the non-negative matrix factorization (NMF) algorithm for heterogeneous high-performance-computing systems. The proposed implementation is based on prior work on NMFk, which can perform automatic model selection and extract latent variables and patterns from data. In this work, we extend NMFk by adding support for dense and sparse matrix operation on multi-node, multi-GPU systems. The resulting algorithm is optimized for out-of-memory problems where the memory required to factorize a given matrix is greater than the available GPU memory. Memory complexity is reduced by batching/tiling strategies, and sparse and dense matrix operations are significantly accelerated with GPU cores (or tensor cores when available). Input/output latency associated with batch copies between host and device is hidden using CUDA streams to overlap data transfers and compute asynchronously, and latency associated with collective communications (both intra-node and inter-node) is reduced using optimized NVIDIA Collective Communication Library (NCCL) based communicators. Benchmark results show significant improvement, from 32X to 76x speedup, with the new implementation using GPUs over the CPU-based NMFk. Good weak scaling was demonstrated on up to 4096 multi-GPU cluster nodes with approximately 25,000 GPUs when decomposing a dense 340 Terabyte-size matrix and an 11 Exabyte-size sparse matrix of density 10^{-6}.

1 Introduction

NMF is a popular unsupervised learning method that extracts sparse and explainable latent features [1], which are often used to reveal explainable low-dimensional hidden structures that represent and classify the elements of the whole dataset [2]. NMF

Extended author information available on the last page of the article

is used in big data analysis, which plays a crucial role in many problems, including human health, cyber security, economic stability, emergency response, and scientific discovery. With the increased accessibility to data and technology, datasets continue to grow in size and complexity. At the same time, the operational value of the information hidden in patterns in such datasets continues to grow in significance. Extracting explainable hidden features from large datasets, collected experimentally or computer-generated, is vital because the data presumably carries essential (but often previously unknown) information about the investigated phenomenon's causality, relationships, and mechanisms. Discovering meaningful hidden patterns from data is not a trivial task because the datasets are formed only by directly observable quantities while the underlying processes or features, in general, remain unobserved, latent, or hidden [3].

Analysis of vast amounts of (usually sparse) data via NMF requires novel distributed approaches for reducing computational complexity, speeding up the computation, and dealing with data storage and data movement challenges. Most NMF computations are matrix-matrix multiplications, which GPU accelerators can speed up. The primary performance and scaling limiting factors in NMF implementations on modern heterogeneous HPC systems are high communication costs due to data movement across different system parts (inter-node and intra-node communications). In various cases, these communication delays exceed the time the actual computations take, resulting in poor performance and poor scalability on large distributed systems.

The growth in data volumes outpacing the improvement in hardware specifications is causing significant challenges in extracting useful information from large-scale datasets using algorithms like NMF. This motivates the need for out-of-memory implementations of NMF for distributed HPC systems, which will allow the decomposition of large datasets that does not fit in memory at once. Enabling out-of-memory factorization is very important because it removes the matrix size constraint imposed by the GPU memory, thus enabling the analysis of datasets up to the cumulative size of all RAM on the cluster. This is mainly required to address the challenges presented by the need to factorize the ever-growing datasets. We utilize this unique ability of pyDNMF-GPU to demonstrate the decomposition of record large dense, and sparse datasets.

To illustrate how *pyDNMF-GPU* can be used as a building block for more comprehensive workflows, we integrate *pyDNMF-GPU* with our existing model selection algorithm *pyDNMFk*[1] that enables automatic determination of the (usually unknown) number of latent features on a large scale datasets [4–8]. We utilized the integrated model selection algorithm previously to decompose the worlds' largest collection of human cancer genomes [9], defining cancer mutational signatures [10], as well as successfully applied to solve real-world problems in various fields [8, 11–19].

This integration results in our out-of-memory scalable tool, *pyDNMFk-GPU*, to be capable of estimating the number of latent features in extra-large *sparse* (tens of

---

[1] *pyDNMFk:* https://github.com/lanl/pyDNMFk.

EBs) and dense (hundreds of TBs) datasets while operating across CPU-GPU hardware. To the best of our knowledge, our framework is the first to identify hidden features in large-scale dense and sparse datasets.

In experiments on large HPC clusters, we show *pyDNMF-GPU*'s potential: we measure up to 76x improvement on a single GPU over running on a single 18-core CPU. We also demonstrate weak scaling on up to 4096 multi-GPU cluster nodes with approximately 25,000 GPUs when decomposing a dense 340 Terabyte-size matrix and an 11 Exabyte-size sparse matrix of density 10^{-6}.

Our main contribution is a novel NMF parallel framework, called *pyDNMF-GPU*, that minimizes the data movement on GPUs, improving overall running times. Our work's main contribution and novelty is the proposal of a new distributed implementation of NMF with low memory complexity that enables the out-of-memory factorization of very large datasets. Our proposed implementation, *pyDNMF-GPU*, takes advantage of the following three modern design choices:

- *pyDNMF-GPU* reduces the latency associated with local data transfer between the GPU and host (and vice-versa) by using *CUDA streams*.
- Latency associated with collective communications (intra-node and inter-node) is reduced by using *NCCL primitives*.[2]
- We incorporate a batching approach for inter-node communication, which provides a unique ability to perform out-of-memory NMF while using multiple GPUs for the bulk of computations.

The main contributions of the paper include:

- Introducing a novel distributed algorithm with out-of-memory support for NMF for sparse and dense matrices operating across CPU-GPU hardware.
- Report, the first NCCL communicator accelerated NMF decomposition tool in distributed GPUs.
- Demonstrate the framework's scalability over a record-breaking 340 Terabytes (TB) dense and 11 Exabytes (EB) sparse synthetic datasets.

The remainder of the paper is organized as follows: Sect. 2 gives a summary of NMF and the existing parallel NMF implementations. In Sect. 3, we detail the design considerations and choices for a scalable, parallel, and efficient algorithm in different configurations of the data size and available GPU VRAM, as well as the complexity of the new implementation. The efficacy of pyDNMF-GPU with different benchmark results and the validation of benchmark results on a synthetic dataset with a predetermined number of latent features is shown in Sect. 4. We finally conclude with summaries and suggestions of possible future work directions in Sect. 5.

---

[2] *NCCL:* https://developer.nvidia.com/nccl.

2 Background and related work

2.1 Non-negative matrix factorization algorithms

NMF [1] approximates the non-negative observational matrix $A \in \mathbb{R}_+^{m \times n}$ with a product of two non-negative factor matrices $W \in \mathbb{R}_+^{m \times k}$ and $H \in \mathbb{R}_+^{k \times n}$ where the columns of W represent the latent features, while the columns of H are the coordinates/weights of the analyzed samples (the columns of A) in the reduced latent space, and k is the latent dimension of the data. The NMF minimization is based on alternating update of each one of these two factor matrices until convergence indicated by the condition $\|A - WH\|_F \leq \eta$ is reached. Here $\|.\|_F$ is the Frobenius norm, $\|A\|_F = \sqrt{\sum_i \sum_j a_{ij}^2}$, where a_{ij} is the element on row i and column j, and η is the desired tolerance. Each iteration consists of a W-update sub-step followed by a H-update sub-step, given by

$$W \xleftarrow[W \geqslant 0]{} \|A - WH\|_F^2$$
$$H \xleftarrow[H \geqslant 0]{} \|A - WH\|_F^2, \tag{1}$$

Algorithm 1 $W, H = \text{NMF}(X, k)$ – Generic *NMF*

Require: $X \in \mathbb{R}_+^{m \times n}$, k is the rank of approximation and max_iters is the number of iterations.
  1: Initialize $W, H = \text{rand}(m, k), \text{rand}(k, n)$
  2: $i = 0$
  3: $\eta_i = \eta + 1$ $\triangleright$ Ensure $\eta_i > \eta$ to enter loop
  4: **while** $(\eta_i \geq \eta \text{ or } i \leq i_{max})$ **do**
      @ stands for matrix multiplication operation
  5:     $W \leftarrow W * \frac{(A@H^T)}{W@H@H+\epsilon}$ $\triangleright$ W update
  6:     $H \leftarrow H * \frac{(W^T@A)}{W^T@W@H+\epsilon}$ $\triangleright$ H update
  7:
  8:     $X \leftarrow WH$
  9:     $\eta_i = \|A - X\|_F^2$
 10:     $i = i + 1$;
 11: **end while**

The Frobenius norm (FRO) based multiplicative update (MU) algorithm is presented in Algorithm 1. In addition to the presented Frobenius norm-based MU algorithm (which leads to a Gaussian model of the noise [20]) other similarities (e.g., KL-divergence that corresponds to a Poisson model) can also be used in the NMF minimization. Also, based on the update rules, several variants of NMF algorithms exist such as hierarchical alternating least squares (HALS) [21], alternating

non-negative least squares with block principle pivoting (ANLS-BPP) [22], and block coordinate descent algorithm (BCD) [23]. These algorithms have different advantages in the context of convergence rate, computational, and memory requirements. MU-based updates are computationally and memory-wise cheap at the cost of slower convergence. Whereas HALS, BCD, and ANLS-BPP have faster convergence rates at the cost of higher computational and memory requirements and high communication costs for parallel implementations. In our experiments, we use the FRO-based MU algorithm to demonstrate record scalability on large datasets due to its lower computation and communication cost, which can easily be modified with another update algorithm or similarity metric.

2.2 Related work on distributed NMF

Several parallel implementations have been proposed to address the computational need of NMF for large datasets involving multiple and repeated matrix-matrix multiplications of several orders in magnitude. The existing parallel implementations can be grouped under two categories (i) with shared memory and (ii) with distributed memory. Majority of existing parallel works utilize shared-memory multiprocessor [24–27] and shared memory GPUs [26–29] via OpenMP and CUDA libraries respectively. A majority of distributed memory implementations rely on *MPI primitives* for distributed CPU [12, 30] and CUDA-aware *MPI primitives* for distributed GPU [28, 30] parallelization. Although shared-memory implementations drastically minimize the communication costs incurred for distributed memory implementation [26], there is a constraint on how much data such frameworks can decompose. Due to this constraint, shared-memory implementation often cannot provide the computational/memory requirements needed for the current large-scale datasets.

Almost all distributed GPU implementations including *NMF-mGPU* [28] and *PLANC* [33] rely on significant data communication for the update of the factors. This involves using CUDA-aware MPI primitives for data communication or MPI distributed memory offload through NVBLAS [33] without multi-node GPU communicators. Such implementation leads to high data movement costs due to data on-loading/offloading to/from the device, which significantly raises communication costs compared to the computation cost for large data decomposition. This is previously illustrated with distributed BPP in *PLANC* [30] and distributed MU and BCD [12] where the communication cost is minimized by communicating only with the two-factor matrices and other partitioned matrices among MPI processes. These works attempt to reduce the bandwidth and data latency using MPI collective communication operations. For distributed CPU implementations, this approach works well as the communication cost is significantly lower compared to the computation cost. However, for GPU implementation, communication cost is higher due to

Table 1 A comparison chart for different GPU based NMF implementations

Framework	GPU	Multi-GPU	Sparse	Out-of-Memory	Remarks
nmf-cuda [24]	✓	–	–	–	Multithreading support only
NMF-mGPU [28]	✓	✓	–	–	Communication inefficient design,entire factor need to be stored for each GPUs
nmfgpu4R [31]	✓	–	–	–	No distributed support.
pytorch-NMF	✓	–	✓	–	No distributed support.
NMF-spark [32]	✓	✓	–	–	Inefficient scaling results.
ALO-NMF [26]	✓	–	✓	–	Shared memory implementation of optimized HALS algorithm and lack of distributed support
PLANC [30]	✓	✓	✓	–	Not a significant gain of GPUs over CPUs and lack of ability to handle large sparse and dense data on GPUs
pyDNMF-GPU (**ours**)	✓	✓	✓	✓	NCCL based efficient implementation with significant speedup of GPUs over CPUs and demonstrated scaling performance over 340TB dense and 11EB sparse data.

device/host data transfer; therefore, communication cost is a limiting factor for parallel performance when using many GPUs.

Table 1 illustrates the comparison against the existing parallel NMF implementations. Further, support for factorization of sparse datasets equally adds value for our new *pyDNMF-GPU* framework. Since many of the extra-large datasets, such as the text corpora, knowledge graph embeddings (and, in general, most of the relational datasets), cyber network activity datasets, and many others, are highly sparse, having sparse decomposition support dramatically reduces the memory and computational requirements which otherwise would be a major bottleneck for the dense implementation. Despite the support for a sparse dataset for shared-memory in *ALO-NMF* and *genten* [26, 27] and for distributed memory in *PLANC* [30], there is no specific solution aiming to address the bottlenecks due to extracted dense factors and their communications for large sparse datasets. Even though the largest sparse datasets may be a few MBs in size, due to their extreme sparsity, decomposing such datasets would be challenging for most existing frameworks as the extracted factors are dense and very large. Even for such a small non-zero valued size, the corresponding dense factors could easily explode and require an expensive communication of dense intermediate terms. However, our batching framework provides a solution by accommodating larger intermediate-dense factors, which have not been addressed previously.

2.3 Rationale for an algorithm for the out-of-memory distributed NMF

In *pyDNMF-GPU*, we use a distributed implementation of NMF that aims at efficiently factorizing matrices of all sizes, even those too big to fit on available memory, in out-of-memory scenarios. To this end, *pyDNMF-GPU* accelerates matrix operations using GPUs on modern heterogeneous systems, provides support for sparse matrix operations to deal with practical data sets which are often sparse, and can partition large problems into smaller problems solved in a distributed manner. Above all, and to the best of our knowledge, our proposed implementation is the first to provide a solution for practical out-of-memory cases that require the factorization of data too big to be stored on combined available GPU memory.

When performing NMF on GPUs, OOM situations can arise in various scenarios with different degrees of complexity. As discussed in [34], we distinguish three main types of OOM scenarios. Scenarios of *type 0* (OOM-0) concern practical problems where the input data A and its co-factors W and H can easily be stored on GPU memory. However, an explosion of memory requirement can occur, either due to the unknown rank k becoming significant, causing W and H to become prohibitively expensive to store on memory, or when computing intermediate results such as $X = WH$ (line 8 of Algorithm 1), when A is a large sparse matrix of very low density, where X resulting from the operation becomes dense and very likely

impossible to store on GPU. For instance, if $A \in \mathbb{R}^{10^6 \times 10^6}$ is a sparse matrix, with density of $\delta \approx 10^{-3}$, the size of A in dense format, in single precision, is $S_A \approx 4TB$, however representing A in CSR sparse format can lower the size of A down to $S_s \sim 3 \times S_A \times \delta \approx 12\ GB$ (the factor of 3 accounts for storing the data, indices and index pointers for CSR format), consequently $S_{NMF} \approx 2 \times S_A \approx 4TB$. Assuming very small k, A and all co-factors can be stored on GPU; however, the calculation of the intermediate product X from $X = WH$ would still require a whopping $\sim 8\ TB$ of GPU memory (line 8 of Algorithm 1), making this scenario an OOM-0 problem.

A more complex OOM scenario, *type 1* (OOM-1), arises in cases where matrix A and at most one of its co-factors cannot be cached on GPU memory; this is typically the case when dealing with a large A that is dense or sparse with high density. Scenarios of *type 2* (OOM-2) are the most complex and consist of practical cases where neither A, nor its co-factors can be stored on GPU memory. Note that more complexity can arise in cases where data cannot fit on host RAM memory, but that still is of *type 2* as the OOM classification here is based on the GPU RAM memory utilization. In other words, in OOM-0 scenarios, all the data can be cached on GPU; in OOM-1 scenarios, the data can partially be cached on GPU, and in OOM-2 scenarios, none of the data can be cached on GPU. The treatment of OOM-2 scenarios is out of the scope of this study.

OOM-0 cases can easily be handled using *tiling* techniques, and OOM-1 cases can be handled with *batching* techniques. In extreme OOM-1 cases, we will complement *batching* by *tiling* to further reduce memory footprint.

Both *batching* and *tiling* are block-based computational techniques designed to simplify larger, memory-intensive computations into smaller, manageable, and partially solvable tasks. Each technique, however, functions in a distinct setting and serves a different purpose. *Batching* is a process that operates on the host, necessitating consistent data transfer between the host and the device. The efficacy of batching techniques is heavily reliant on the speed of the interconnecting buses between the host and device, such as PCIe or NV-Link. *Batching* techniques become crucial when dealing with OOM-1 problems, as they help in transferring partially computed results. Conversely, *tiling* happens directly within the device memory, resulting in data transfer between global memory and shared or cache memory. The performance of *tiling* techniques is primarily governed by the GPU architecture, including features like memory speed and available shared memory. *Tiling* techniques are especially effective for tackling OOM-0 problems, as they handle computational tasks directly on the device. Notably, *batching* is typically irrelevant for OOM-0 problems as these computations are already based on the device. Similarly, *tiling* techniques alone cannot address OOM-1 issues due to the preliminary need to transfer operands to the device. However, an optimized solution for extreme OOM-1 problems can be achieved by strategically combining both *batching* and *tiling* techniques, thus enhancing the overall performance.

In the section below, we discuss our implementation and design choices.

3 *pyDNMF-GPU* for heterogeneous systems

An efficient implementation of NMF for distributed heterogeneous systems should avoid high costs associated with communication (data transfer) resulting from poor consideration for data locality in the distribution of the computational work. Furthermore, cases, where resources such as available combined GPUs memory are limited will require additional considerations and various trade-offs. For instance, it is sometimes better to replicate data over the distributed compute grid to reduce communication. Other times, it is acceptable to use batching techniques that can increase communication costs to lower the memory footprint. Below we first discuss our distributed data partition strategies that partition large problems into smaller problems solvable on cooperative distributed systems in subsection 3.1, and then in subsection 3.2 we discuss our tiling and batching approaches, respectively used to handle practical scenarios of complexities *type*0 and *type*1.

3.1 Distributed implementation

Our implementation considers two one-dimensional data partition strategies based on the shape of matrix A ($m \times n$). A column (vertical) partition, CNMF employed when $n > m$, and a row (horizontal) partition, RNMF, is used otherwise.

Algorithm 2 $\boldsymbol{W}, \boldsymbol{H} = \text{DCNMF}(\boldsymbol{X}, k)$–*Distributed CNMF Algorithm*

Require: $\boldsymbol{X} \in \mathbb{R}_+^{m \times n}$, k is the rank of approximation and max_iters is the number of iterations.

Require: $\boldsymbol{X}$ distributed across N GPUs where $\boldsymbol{X}_j \in \mathbb{R}^{m \times n/N}$ where $J = n/N$ if $m < n$. Similarly the co-factor $\boldsymbol{W}$ local to each GPU given by $\boldsymbol{W} \in \mathbb{R}^{m \times k}$ which is reproduced across different GPUs. $\boldsymbol{H}$ is distributed across N GPUS such that $\boldsymbol{H}_j \in \mathbb{R}^{k \times J}$.

1: Initialize $\boldsymbol{W}, \boldsymbol{H}_j = \text{rand}(m, k), \text{rand}(k, J)$
   $J = n/N, j_0 = gID * J, j_1 = (gID + 1) * J$
2: **for** $l = 0$ to $l < max\_iter$ **do**
3:     $\boldsymbol{WTA} = \boldsymbol{W}^{(l)^T}@\boldsymbol{A}$ ▷ @ stands for matrix multiplication operation
4:     $\boldsymbol{WTW} = \boldsymbol{W}^{(l)^T}@\boldsymbol{W}$
5:     $\boldsymbol{H}^{(l+1)} = (\boldsymbol{H}^{(l)} * \boldsymbol{WTA})/(\boldsymbol{WTW}@\boldsymbol{H}^{(l)} + \epsilon)$ ▷ H_{update}
6:     $\boldsymbol{HHT} = \boldsymbol{H}^{(l+1)}@\boldsymbol{H}^{T^{(l+1)}}$
7:     $\boldsymbol{HHT} = \text{All\_Reduce}(\boldsymbol{HHT})$
8:     $\boldsymbol{WHHT} = \boldsymbol{W}^{(l)}@\boldsymbol{HHT}$
9:     $\boldsymbol{AHT} = \boldsymbol{A}@\boldsymbol{HT}$
10:     $\boldsymbol{AHT} = \text{All\_Reduce}(\boldsymbol{AHT})$
11:     $\boldsymbol{W}^{(l+1)} = \boldsymbol{W}^{(l)} * \boldsymbol{AHT}/(\boldsymbol{WHHT} + \epsilon)$ ▷ W_{update}
12: **end for**

(a) Column partition with *orthogonal* batching

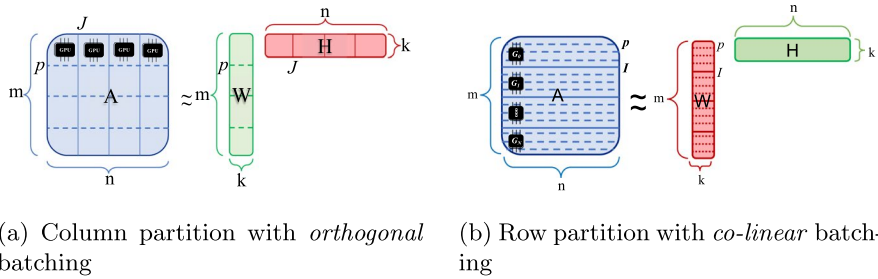(b) Row partition with *co-linear* batching

Fig. 1 Illustration of distributed matrix **A** and co-factors **W** and **H** in *CNMF* and *RNMF* distributed partitions respectively in (**a**) and (**b**). Solid lines show distributed partition boundaries, and dashed lines show local partition segmentation in batch for Out-of-Memory decomposition

---

Algorithm 3 $W, H = \mathrm{DRNMF}(X, k)$–*Distributed RNMF Algorithm*

Require: $X \in \mathbb{R}_+^{m \times n}$, k is the rank of approximation and max_iters is the number of iterations.

Require: X distributed across N GPUs where $X_i \in \mathbb{R}^{m/N \times n}$ where $I = m/N$ if $n < m$. Similarly the co-factor H is reproduced across different GPUs given by $H \in \mathbb{R}^{k \times n}$. W is distributed across N GPUS such that $W_i \in \mathbb{R}^{I \times k}$.

1: Initialize $W_i, H = \mathrm{rand}(I, k), \mathrm{rand}(k, n)$
   $I = m/N, i_0 = gID * I, i_1 = (gID + 1) * I$
2: **for** $l = 0$ to $l < max\_iter$ **do**
3:     $WTA = W^{(l)^T}@A$ ▷ @ stands for matrix multiplication operation
4:     $WTA = \mathrm{All\_Reduce}(WTA)$
5:     $WTW = W^{(l)^T}@W$
6:     $WTW = \mathrm{All\_Reduce}(WTW)$
7:     $H^{(l+1)} = (H^{(l)} * WTA)/(WTW@H^{(l)} + \epsilon)$ ▷ H_{update}
8:     $HHT = H^{(l+1)}@H^{T^{(l+1)}}$
9:     $WHHT = W^{(l)}@HHT$
10:    $AHT = A@HT$
11:    $W^{(l+1)} = W^{(l)} * AHT/(WHHT + \epsilon)$ ▷ W_{update}
12: **end for**

Assuming a distributed system with *N* GPUs where each GPU is indexed by its global rank g_{ID}. In the *CNMF* approach illustrated in Fig. 1a, the jth GPU with $g_{ID} = j$ will work on array partitions $A[:, j_0 : j_1]$, $H[:, j_0 : j_1]$ and W, where $j_0 = j \times J$, $j_1 = (j + 1) \times q$, and $J = n/N$(partition size). Each GPU gets a full copy of *W* (*W* is replicated) and a unique partition of *A* and *H*. This translates into a segmentation of arrays *A* and *H* on global memory illustrated with solid lines in Fig. 1a. These solid lines indicate boundaries in global memory
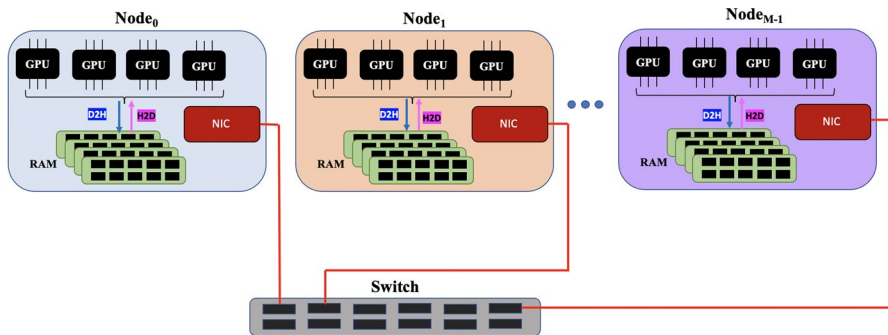
Fig. 2 Illustration of distributed HPC hardware and different communication channels

and consequently help conceptualize where communication is required whenever information is exchanged from one bounded region to another. The *H*-update is embarrassingly parallel since W^TW, $(W^TW)H$, and W^TA can all be computed locally on each GPU; the *W*-update on the other hand, will require two separate all-reduce-sum communications to compute AH^T and HH^T as indicated in Algorithm 2 lines 10 and 7.

Following a similar analogy, a *RNMF* approach results with **H** replicated on the different GPUs and **A** and **H** distributed across the compute grid. This time **W**-update is embarrassingly parallel since HH^T, $W(HH^T)$, and AH^T can all be computed locally on each GPU, but the *H*-update will require separate all-reduce-sum communication to compute W^TW and W^TA as presented in Algorithm 3.

Communication takes place through various channels with different bandwidths and latency. We refer to intra-node communications as any communication on the same node, i.e., yellow, pink, and black lines in Fig. 2 and those between different nodes as inter-node communications. i.e red lines in Fig. 2. The latter often have the lowest bandwidth and highest latency and could easily cause bottlenecks for distributed algorithms such as NMF. For these practical reasons, in our implementation, we avoid all-reduce collective calls as much as possible. When $n > m$, *CNMF* is more efficient than *RNMF* because it costs less to communicate AH^T of shape $m \times k$, and *RNMF* is more efficient when $m > n$ because it cost less to communicate W^TA of shape $k \times n$.

The FLOP (floating point operations) count for the given distributed RNMF (row-wise nonnegative matrix factorization) algorithm can be calculated by going through each of the operations performed in the algorithm. Below is a rough estimation of the FLOP count for each line of interest in the algorithm:

- Matrix multiplication (Line 3): $WTA = [l][] W^T @ A$. Here we have a matrix multiplication of size $(k \times I) * (I \times n)$, which will result in $2k * I * n - k * n$ FLOPs.
- Matrix Multiplication (Line 5): $WTW = [l][] W^T @ W$. Here we have a matrix multiplication of size $(k \times I) * (I \times k)$, which will result in $2k * I * k - k * k$ FLOPs.
- Elementwise Multiplication and Division (Line 7): $[l+1][]H = ([l][]H * WTA)/(WTW @ [l]H + \epsilon)$. This consists of $k * n$ FLOPs for elementwise multiplication and $k * n$ FLOPs for elementwise division, so total $2 * k * n$ FLOPs.
- Matrix multiplication (Line 8): $HHT = [l+1][] H @ H^{T^{(l+1)}}$. Here we have a matrix multiplication of size $(k \times n) * (n \times k)$, which will result in $2k * n * k - k * k$ FLOPs.
- Matrix multiplication (Line 9): $WHHT = [l] W @ HHT$. Here we have a matrix multiplication of size $(I \times k) * (k \times k)$, which will result in $2I * k * k - I * k$ FLOPs.
- Matrix multiplication (Line 10): $AHT = A @ HT$. Here we have a matrix multiplication of size $(I \times n) * (n \times k)$, which will result in $2I * n * k - I * k$ FLOPs.
- Elementwise multiplication and division (Line 11): $[l+1][]W = [l][]W * AHT/(WHHT + \epsilon)$. This consists of $I * k$ FLOPs for elementwise multiplication and $I * k$ FLOPs for elementwise division, so total $2 * I * k$ FLOPs.

Note: The All_Reduce operation (Lines 4 and 6) are communication operations and are not considered in the FLOP count as they do not involve any computation.

So, total FLOPs for each iteration of the loop = $2k * I * n + 2k * I * k + 2 * k * n$ $+2k * n * k + 2I * k * k + 2I * n * k + 2 * I * k - k * n - k * k - k * k - I * k - I * k$.

For max_{iter} iterations, total FLOPs would be max_{iter} times the FLOPs per iteration. Now, to compute GFLOPS, we have GFLOPS = total_FLOPs/(total_time×1e9). Morever, given device peak GFLOPS (peakG), we can compute efficiency as GFLOPS/peakG*100%.

The total VRAM required to factorize A of size $size(A) = S_A$ (in Bytes) is typically in the order of $S_{NMF} \sim 4 \times S_A$. One fold of S_A to store A in memory, another fold to store perturbed A [7], an additional fold to compute intermediate product $X = W @ H$ when checking the convergence condition $\| A - WH \|_F \leq \eta$, and almost one full fold to store the co-factors W, H, and heavy intermediate products such as $W^T A$ or AH^T. When the total available combined GPU VRAM, S_{GV}, is lower than S_{NMF}, as in practical big data applications, batching techniques are imperative. Batching, in most cases, increases intra-node and inter-node communication overheads. Although this can significantly affect the algorithm's performance, proper use of asynchronous data copy and CUDA streams can reduce performance loss by overlapping compute and data transfers, as discussed in our out-of-memory implementation below.

3.2 Out-of-memory implementation and memory complexity analysis

---

Algorithm 4 $W, H = \text{CNMF}(X, k)$–*CNMF* with *orthogonal* batching

---

Require: $X \in \mathbb{R}_+^{m \times n}$, k is the rank of approximation and max_iters is the number of iterations.

Require: X distributed across N GPUs where $X_j \in \mathbb{R}^{m \times n/N}$ where $J = \frac{n}{N}$ if $m < n$. Locally to each GPU, X_j can be split into n_b batches where $n_b = \frac{m}{p}$ following a *orthogonal* batching strategy where the batch is of shape $p \times J$. Similarly the co-factor W local to each GPU given by $W \in \mathbb{R}^{m \times k}$ is locally divided into n_b batches where the batch is of shape $p \times k$. H is distributed across N GPUS such that $H_j \in \mathbb{R}^{k \times J}$.

1: Initialize $W_i, H_j = \text{rand}(I, k), \text{rand}(k, J)$
2: Initialize $\mathbf{SQ}$, a queue of CUDA-streams of size q_s.
3: Initalize zero array accumulators $\mathcal{WTA} \in \mathbb{Z}^{q_s \times k \times n}$ and $\mathcal{WTW} \in \mathbb{Z}^{q_s \times k \times k}$
4: **for** l in [1,max_iters] **do**
/* **Update H given W** */
5:     **for** b in n_b **do**
6:         $\mathbf{SQ} -> st$ $\triangleright$ De-queue stream st from $\mathbf{SQ}$
7:         **while** in context st **do** $\triangleright$ Calculations in loop are in non-default stream st
    $i_0 = b * p, \; i_1 = (b+1) * p, \; j_0 = gID * J, \; j_1 = (gID + 1) * J$
8:             $A = \text{H2D}(X_j[i0:i1,:], st)$ $\triangleright$ H2D(x, st) stands for async copy of x from Host to GPU using non-default stream st
9:             $W_b = \text{H2D}(W^{(l)}[i0:i1,:], st)$
10:             $\mathcal{WTA}[st] \mathrel{+}= W_b^T @ A$ $\triangleright$ Accumulate local $W^T @ A$
11:             $\mathcal{WTW}[st] \mathrel{+}= W_b^T W_b$ $\triangleright$ Accumulate local $W^T @ W_b$
12:             $st -> \mathbf{SQ}$ $\triangleright$ En-queue stream st back into $\mathbf{SQ}$, exit context st
13:         **end while**
14:     **end for**
15:     $WTA = \sum_{i=1}^{n_b} \mathcal{WTA}_{i,:,:}$ $\triangleright$ Reduce of $\mathcal{WTA}$ local to each GPU
16:     $WTW = \sum_{i=1}^{n_b} \mathcal{WTW}_{i,:,:}$ $\triangleright$ Reduce of $\mathcal{WTW}$ local to each GPU
17:     $H_j^{(l+1)} \times = WTA / (WTW @ H_j^{(l)} + \epsilon)$
/* **Update W given W** */
18:     $HHT = H_j^{(l+1)} @ H_j^{T^{(l+1)}}$
19:     $HHT \leftarrow \text{All\_Reduce\_sum}(HHT)$
20:     **for** b in 0 to n_B **do**
21:         $\mathbf{SQ} -> st$ $\triangleright$ De-queue stream st from $\mathbf{SQ}$
22:         **while** in context st **do** $\triangleright$ Calculations in loop are in non-default stream st
23:             Set $i_0 = b * p, \; i_1 = (b+1) * p$
24:             $A = \text{H2D}(X_j[i0:i1,:], st)$ $\triangleright$ H2D(x, st) stands for async copy of x from Host to GPU using non-default stream st
25:             $W_b = \text{H2D}(W^{(l)}[i0:i1,:], st)$
26:             $WHHT_b = W_b @ HHT + \epsilon$
27:             $AHT = A @ H_j^{T^{(l+1)}}$
28:             $AHT = \text{All\_Reduce\_sum}(AHT, stream = st)$ $\triangleright$ Perform stream aware Reduce with NCCL
29:             $W_b \times = AHT / (WHHT_b + \epsilon)$
30:             $W_i^{(l+1)}[i0:i1,:] = \text{D2H}(W_b, st)$ $\triangleright$ D2H(x, st) stands for async copy of x from GPU to Host using non-default stream st.
31:         **end while**
32:     **end for**
33: **end for**

In *pyDNMF-GPU*, *OOM-0* problems are handled using a tiling approach where temporary results like AH^T, W^TA or WH are evaluated in small chunks, by tiling one of the operands, such that the size of the tile sets the memory required for the calculation. In RNMF for instance, the criterion $\|A - WH\|_F \leq \eta$, can be evaluated in m/p small chunks obtained by tiling W into smaller tiles of size $p \times k$. This results in computing nt chunks of $[\|A - WH\|_F]_p$ which are accumulated into the total error e such that $e = \sum ([\|A - WH\|_F]_t)_{t=0}^{m/p-1}$, which can later be used to check the conversion condition $e \leq \eta$. This allows the reduction of the memory required to check the conversion criterion from $\mathcal{O}(m \times n)$ to $\mathcal{O}(p \times n)$. Because all matrices involved in the calculations are stored on GPU memory, performance loss due to tiling can be negligible, especially on modern GPU architecture like NVIDIA Ampere A100, which uses low latency and high bandwidth HBM memory. Using the tiling approach, the memory required to perform NMF on GPU can be reduced from $S_{NMF} \sim 4S_A$ to approximately $2S_A \leq S_{NMF} \leq 3S_A$.

Algorithm 5 $\boldsymbol{W}, \boldsymbol{H} = \text{RNMF}(\boldsymbol{X}, k)$–*RNMF* with *co-linear* batching

---

Require: $\boldsymbol{X} \in \mathbb{R}_+^{m \times n}$, k is the rank of approximation and max_iters is the number of iterations.

Require: $\boldsymbol{X}$ distributed across N GPUs where $\boldsymbol{X}_i \in \mathbb{R}^{I \times n}$ where $I = \frac{m}{N}$ if $m > n$. Locally to each GPU, $\boldsymbol{X}_i$ can be split into n_b batches following a *co-linear* batching strategy where the batch size is $b_s = \frac{I}{n_b} \times n$. Similarly the co-factor $\boldsymbol{W}$ local to each GPU given by $\boldsymbol{W}_i \in \mathbb{R}^{I \times k}$ is locally divided into n_b batches where $b_s = \frac{I}{n_b} \times k$. $\boldsymbol{H} \in \mathbb{R}^{k \times n}$ is cached and replicated across the GPUs.

1: Initialize $\boldsymbol{W}_i, \boldsymbol{H} = \text{rand}(I, k), \text{rand}(k, n)$
2: Initialize $\mathbf{SQ}$, a queue of CUDA-streams of size q_s.
3: Initalize accumulators $\boldsymbol{\mathcal{WTA}} \in \mathbb{R}_+^{q_s \times k \times n}$ and $\boldsymbol{\mathcal{WTW}} \in \mathbb{R}_+^{q_s \times k \times k}$
4: **for** l in $[1, max\_iters]$ **do**
    /* **Update $\boldsymbol{W}$ given $\boldsymbol{H}$** */
5:    $\boldsymbol{HHT} = \boldsymbol{H}^{(l)} @ \boldsymbol{H}^{T^{(l)}}$
6:    **for** p in n_b **do**
7:       $io, i_1 = p * b_s, (p+1) * b_s$
8:       $\mathbf{SQ} -> st$ $\triangleright$ De-queue stream st from $\mathbf{SQ}$
9:       **while** in context st **do** $\triangleright$ Calculations in loop are in non-default stream st
10:          $\boldsymbol{A} = \text{H2D}(\boldsymbol{X}_i[i0:i1,:], st)$ $\triangleright$ H2D($\boldsymbol{x}, st$) stands for async copy of $\boldsymbol{x}$ from Host to GPU using non-default stream st
11:          $\boldsymbol{W}_b = \text{H2D}(\boldsymbol{W}_i^{(l)}[i0:i1,:], st)$
12:          $\boldsymbol{AHT} = \boldsymbol{A} @ \boldsymbol{H}^{T^{(l)}}$
13:          $\boldsymbol{WHHT} = \boldsymbol{W}_b @ \boldsymbol{HHT} + \epsilon$
14:          $\boldsymbol{W}_b \times = \boldsymbol{AHT} / \boldsymbol{AWWT}$ $\triangleright$ $\boldsymbol{W}$ update
15:          $\boldsymbol{W}_i^{(l+1)}[i0:i1,:] = \text{D2H}(\boldsymbol{W}_b, st)$ $\triangleright$ D2H($\boldsymbol{x}, st$) stands for async copy of $\boldsymbol{x}$ from GPU to Host using non-default stream st.
16:          $\boldsymbol{\mathcal{WTA}}[st] += \boldsymbol{W}_b^T @ \boldsymbol{A}$ $\triangleright$ Accumulate local $\boldsymbol{W}^T @ \boldsymbol{A}$
17:          $\boldsymbol{\mathcal{WTW}}[st] += \boldsymbol{W}_b^T \boldsymbol{W}_b$ $\triangleright$ Accumulate local $\boldsymbol{W}^T @ \boldsymbol{W}_b$
18:          $st -> \mathbf{SQ}$ $\triangleright$ En-queue stream st back into $\mathbf{SQ}$, exit context st
19:       **end while**
20:    **end for**
    /* **Update $\boldsymbol{H}$ given $\boldsymbol{W}$** */
21:    $\boldsymbol{WTA} = \sum_{i=1}^{n_b} \boldsymbol{\mathcal{WTA}}_{i,:,:}$ $\triangleright$ Reduce of $\boldsymbol{\mathcal{WTA}}$ local to each GPU
22:    $\boldsymbol{WTW} = \sum_{i=1}^{n_b} \boldsymbol{\mathcal{WTW}}_{i,:,:}$ $\triangleright$ Reduce of $\boldsymbol{\mathcal{WTW}}$ local to each GPU
23:    $\boldsymbol{WTA} = \text{All\_Reduce\_sum}(\boldsymbol{WTA})$ $\triangleright$ Global Reduce of $\boldsymbol{WTA}$ across all GPUs
24:    $\boldsymbol{WTW} = \text{All\_Reduce\_sum}(\boldsymbol{WTW})$ $\triangleright$ Global Reduce of $\boldsymbol{WTW}$ across all GPUs
25:    $\boldsymbol{WTWH} = \boldsymbol{WTW} @ \boldsymbol{H}^{(l)}$
26:    $\boldsymbol{WTWH} = \boldsymbol{WTW} @ \boldsymbol{H}^{(l)} + \epsilon$
27:    $\boldsymbol{H}^{(l+1)} \times = \boldsymbol{WTA} / \boldsymbol{WTWH}$ $\triangleright$ $\boldsymbol{H}$ update
28: **end for**

When dealing with *OOM-1* cases, light arrays are cached on GPU memory, and heavier arrays are kept on host memory and batched to respective GPUs as needed. Further, an appropriate batching strategy for the chosen memory partition is required to limit unnecessary *D2H* and *H2D* copies. In *PyDNMFk-GPU*, we employ a *1D co-linear* batching strategy, illustrated in Fig. 1b, where the elements in the batch are arrays of length equal *max(m, n)*. This batching strategy turns out to employ half the *D2H* and *H2D* memory copies required by an *orthogonal* batching
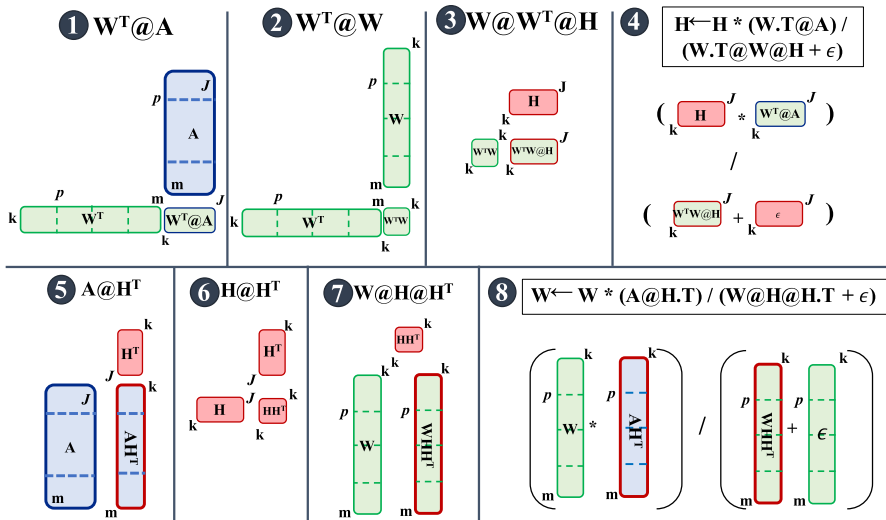
Fig. 3 Illustration of the batched multiplicative update of Algorithm 4 for the column partition(CNMF). Green array is duplicated across different MPI ranks. Blue and red arrays are distributed, and only red array is cached on device. For CNMF, p is out-of-memory batch width and J is distributed partition width

strategy, illustrated in Fig. 1a for the column partition, where the elements in the batch are vectors of length equal $min(m, n)$. Let p be a batch size control parameter. In *RNMF* (*CNMF*) the number of batches is then given by $n_B = m/p$ ($n_B = n/p$). In the extreme case where both m and n are very large, only the light array, $W[J, :]$ is cached on GPU memory, and heavier arrays $A[J, b_0 : b_i]$ ($A[b_0 : b_1, J]$) and $H[:, b_0 : b_1]$ ($W[b_0 : b_1, :]$) batched to their respective GPUs, such that for the bth batch, $b_0 = b \times p$ and $b_1 = (b + 1) \times p$.

An implementation of the distributed *CNMF* with *orthogonal* batching is given in Algorithm 4. The calculation of the different intermediate products is illustrated in Fig. 3, where batch delimitation is represented with dashed lines. The top row shows all intermediate products computed during H-update, and products computed in W-update is shown in the bottom row. Intermediate products $W^T@A$ and $W@W^T$ can be computed with n_B independent batches each containing $[W^T@A]_b$ and $[W^T@W]_b$ sub-products. Each batch is queued to a non-default CUDA stream Stm_b along with the transfer of $A_b[b_0 : b_i, J]$ and $W_b[b_0 : b_i, :]$, and when calculated, each sub-product is added to a local accumulator (see lines 10–11 of Algorithm 4). Once all batches have been processed, all accumulators are reduced to obtain the full values of W^{TW} and W^{TA}, (see lines 15–16 of Algorithm 4). Note that this reduction is local to each GPU and does not involve communication. Special batch en-queuing and de-queuing policies are implemented with CUDA events, so as to limit (control) the number of concurrent batches on GPU to q_s (see lines 6–7,12 of Algorithm 4). This way, the memory requirement for H_{update} is bounded by $q_s \times [p \times J]$, as $W^TW@H$ and $H * (W^TA)/(W^TWH + \epsilon)$ have

a $k \times J$ memory requirement. This is important, especially when dealing with large sparse arrays, which can be cheap to cache on the device but can also have co-factors becoming prohibitively expensive to cache when k becomes large. For instance, in *CNMF*, when $m \sim 10$ *million*, the size of H will approximate $20GB$ in single precision when $k \sim 512$.

Intermediate products $A@H^T$ and $W@HH^T$ of the W-update are computed similarly to $W^T@A$ and $W@W^T$, except $A@H^T$ will require an intermediate $all-reduce-sum$ of sub-products $[A@H^T]_b$ of batches of same stream number from the different GPUs (see line 28 of Algorithm 4). The resulting memory complexity of this implementation is found to be of the order of $\mathcal{O}(p \times n \times q_s)$ when $p >> k$ which is the aggregated memory utilization caused by the q_s concurrent uploads of batches of A of size $p \times n$ at line 8 or line 24 of Algorithm 4. This is a significant saving compared to the estimated $S_{NMF} \sim 3 \times S_A$ when not checking the convergence condition $\|A-WH\|_F \leq \eta$. When the convergence criterion is checked, the error computation is tiled similarly as it was done for *OOM-0* scenarios, resulting in a memory utilisation $S_{NMF} \sim 2 \times p \times n \times q_s$ when $p >> k$.

Note that the use of batches here will only increase $intra-node$ communication due to mem-copies, as it is not possible to cache A and W on the device, however major shortcomings of using the *orthogonal* batching can be pointed out through the example of Algorithm 4 discussed above. First, the need to upload batches two times at lines (8–9 and lines 24-25of Algorithm 4) is very inefficient as the second set of H2D will significantly (almost double) data transfer costs. Second, unnecessary additional latency due to load balancing delays when the streams are scheduled in a different order on the different GPUs can occur at line 28 of Algorithm 4. Above all, the worst result here is that both inefficiencies multiply with the number of iterations (see line 4 of Algorithm 4).

A better implementation uses a *co-linear* batching strategy as it is done in the batched implementation of the distributed *RNMF* given in Algorithm 5. The calculation of the different intermediate products is illustrated in Fig. 4. The top row shows all intermediate products computed during W-update, and products computed in H-update is shown in the bottom row. The W-update (cartoons 1-4 of Fig. 4 is embarrassingly parallel and can be done at a batch level. This means that within each batch, we have the updated partition of W readily available to compute local sub-products $W^T@A$ and $W^T@A$ in the H-update. This avoids the need for a second data upload, as was the case with implementation using an *orthogonal* batching strategy. Further, the aggregation of $W^T@A$ and $W^T@A$ first consists of a local accumulation of the sub-products (lines 16–17 of Algorithm 5) followed by a local reduction (lines 21–22 of Algorithm 5), then a global reduction (lines 23–24 of Algorithm 5) illustrated in cartoons 5–6 of Fig. 4. This does not require communication between batches of the same stream number and consequently avoids load balancing issues as discussed above in case using an *orthogonal* batching strategy.
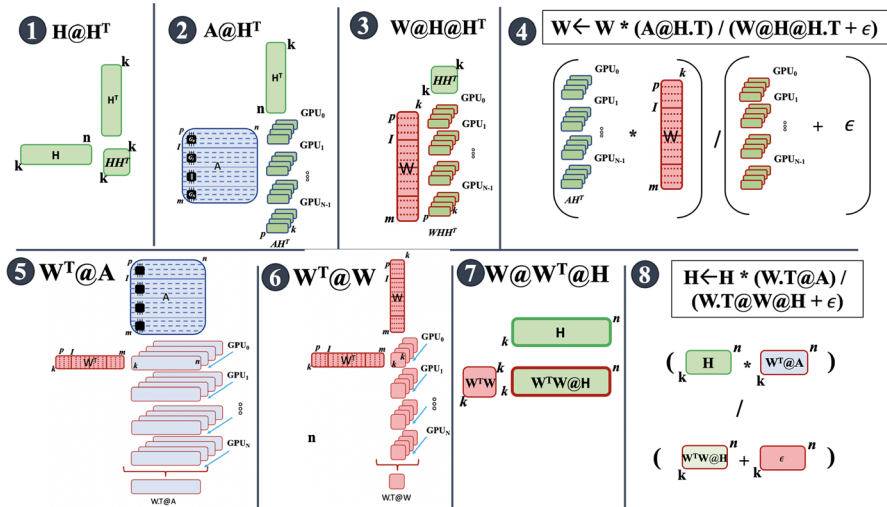
Fig. 4 Illustration of the batched multiplicative update Algorithm 5 for the row partition(RNMF) and colinear batching. Green array is duplicated across different MPI ranks. Blue and red arrays are distributed, and only red array is cached on device. For RNMF, p is out-of-memory batch width and J is distributed partition width

4 Benchmarks results and discussion

4.1 Hardware infrastructure and software environment

Benchmark tests were performed on three different HPC clusters to illustrate the portability and scalability of *pyDNMF-GPU*. The first cluster, *Kodiak*, is a LANL internal HPC cluster with 133 compute nodes with dual Xeon E5-2695 v4 CPUs and four NVIDIA Pascal P100 GPGPUs each. Each NVIDIA Pascal P100 GPGPU has 16GB VRAM and uses PCI-E 16X gen 3 Links. The cluster peaks at 1850TF/s and uses an Infiniband interconnect. Each GPU peaks at 9.3 teraflops for single precision. The second cluster, *Chicoma*, is also a LANL internal HPC cluster, composed of 118 compute nodes where each node has 2 AMD EPYC 7713 Processors and 4 NVIDIA Ampere A100 GPUs. The AMD EPYC 7713 CPUs have 64 cores peaking at 3.67 GHz and 256 GB RAM. Each of the four NVIDIA A100 GPUs in each node provides a theoretical double-precision arithmetic capability of approximately 19.5 teraflops with 40GB VRAM memory. The nodes are networked with HPE/Cray slingshot 10 interconnect with 100Gbit/s bandwidth. *Chicoma* runs Shasta 1.4 OS and SLURM Job manager. The third cluster, *Summit*, peaks at over 200 petaflops in double-precision theoretical performance and comprises 4600 IBM AC922 compute nodes, with two IBM POWER9 CPUs and six NVIDIA Volta V100 GPUs each which peak at 15.7 single precision. The POWER9 CPUs have 22 cores running at 3.07 GHz. The six NVIDIA Tesla V100 GPUs in each node provide a theoretical double-precision arithmetic capability of approximately 40 teraflops with VRAM memory of 16GB/GPU. Dual NVLink 2.0 connections between CPUs and

GPUs provide a 25-GB/s transfer rate in each direction on each NVLink, yielding an aggregate bidirectional bandwidth of 100 GB/s. The nodes are networked in a non-blocking fat-tree topology by Infiniband. Summit deploys an RHEL 7.4 OS and IBM Job step manager jsrun to run compute jobs. Jsrun provides a fine control of how node-level resources are allocated on these systems, including CPU cores, GPUs, and hardware threads.

pyDNMF-GPU is written in python and uses other off the shelf python libraries such as CuPy [35], Numpy [36], MPI4PY [37] and Scipy [38]. It supports dense and sparse datasets on various hardware architectures and handles communication using a low-latency NCCL-based communicator. NCCL is an open-source library providing inter-GPU communication primitives developed and maintained by NVIDIA. NCCL performs automatic hardware topology detection, which it then uses in graph search algorithms to identify communication paths that offer the highest bandwidth and lowest latencies for communication between GPUs intra- and inter-node (e.g., between GPUs that are on the same compute node, as well as between GPUs that are on separate compute nodes). NCCL is compatible with many multi-GPU parallelization models, and provides the ability to perform MPI-like collective and point-to-point operations such as allgather, reduce, broadcast, allreduce, send, and recv. NCCL was initially proposed to help with the need to transfer large message GPU buffers in deep learning applications efficiently. Many leading deep learning frameworks like Chainer, PyTorch, and TensorFlow have since integrated NCCL to accelerate deep learning training on multi-GPU, and multi-node systems, which has motivated us to use NCCL to handle communication in our work. All implementations discussed in the section above were found to benefit from a reduction in data transfer latency and communication performance (both intra-node and inter-node communications), using our low latency NCCL-based communicators versus MPI. An example of such benefit in communication performance gain is illustrated in the subsection 4.2 below that compares the new NMF implementation proposed in this work that uses an NCCL-based communicator to the prior *pyDNMFk* that uses a traditional MPI based communicator. A More comprehensive and detailed comparative study between NCCL and MPI can be found in the analysis by Awan [39].

4.2 Performance benchmark results of *pyDNMF-GPU* vs *pyDNMFk*

The performance gained using GPU over CPU is assessed with speedup computed as the ratio of time measured on CPU with *pyDNMFk* [7], to time measured on GPU with *pyDNMF-GPU*. For this study, we used a dense matrix of shape and size S_A of memory (in bytes) that respectively scale as $[N \times 65536, 32768]$ and $N \times 8GB$, where N is the number of GPU or CPU units. Speedup measured on the *Kodiak* cluster are reported in Fig. 5. Figure 5a shows speedup in NMF time as a function of the number of units for various k. First, we note an increasing speedup with the increasing number of units, and second, we note a decreasing performance with increasing k when $k \geq 32$. The low performance observed at $k < 32$ is explained by low GPU occupancy. The best performance is obtained when $k = 32$, peaking at $76X$. We also report speedup in communication time
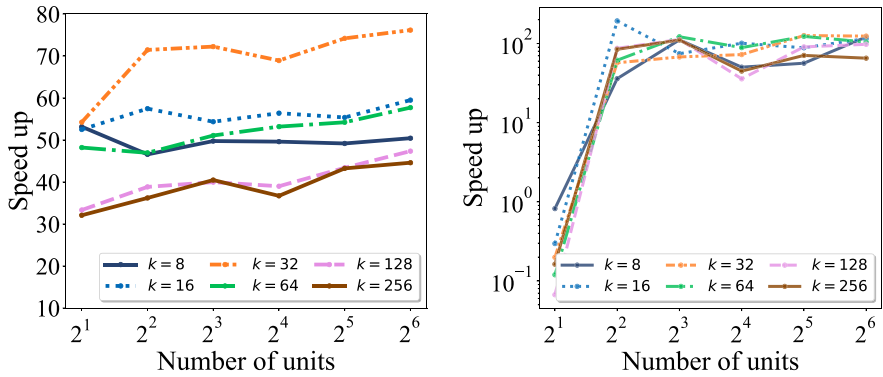
(a) Speedups of the compute times for various latent dimensions on different numbers of compute units.

(b) Speedups of the communication times for various latent dimensions on different numbers of compute units.

Fig. 5 Results of benchmarking experiment showing speedup gain using N GPUs vs N CPUs, for various k. Speedup gained on NMF calculation time is shown in Fig. 5a and speedup gained on communication time is shown in Fig. 5b

computed as the ratio of total communication time measured with *pyDNMFk* to the total communication time measured *pyDNMF-GPU*. The former used MPI based communicator and the latter used an NCCL-based communicator. Speedup in communication time is reported as a function of number of units for various k in Fig. 5b. We note $\sim 80X - 100X$ speedup when $N > 2$, the number of units above which inter-node communications start. This clearly shows a significant performance gain in communication when using NCCL in *pyDNMF-GPU* over MPI in *pyDNMFk*.

4.3 Strong and Weak scalability of *pyDNMF-GPU*

The scalability of the proposed NMFk algorithm is assessed using both strong and weak scaling analysis. This scaling study measures NMF execution time for a given problem size as a function of the number of compute units. Compute nodes (with 4 GPUs each) are chosen as compute units in strong scaling analysis, while individual GPUs are chosen as compute units in weak scaling analysis. The problem size S_A is chosen to use most of the available 16GB VRAM per GPU. To this end, S_A is fixed at $S_A \approx 4 \times 8GB = 32GB$ in strong scaling analysis and chosen to scale as $S_A \approx 8GB \times N$ in weak scaling analysis. This is accomplished by generating a random synthetic array A of shape $[4 \times 65536, 32768]$ and $[N \times 65536, 32768]$ respectively in both strong and weak scaling. Cases of sparse A with density 10^{-5} were also studied, and for those cases, A was generated as a random synthetic array of shape $[4 \times 2097152, 65536]$ in strong scaling analysis, and of shape $[N \times 2097152, 65536]$ was chosen in weak scaling analysis.
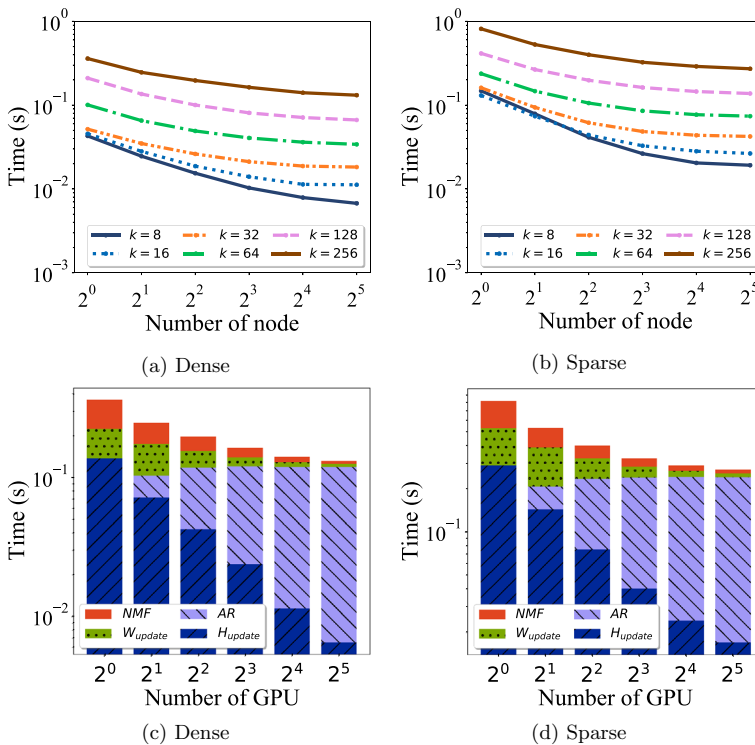
Fig. 6 Results of strong scaling study performed on *Kodiak*. NMF time vs number of node for various *k* dense and sparse *A* respectively shown in (6a) and (6b). For the case $k = 8$, execution time of H_{update}, W_{update} and All-reduce communication are compared in (6c) and (6d), respectively for dense and sparse *A*

4.3.1 Strong scalability

Strong scaling results for cases where $k = 8, 16, 32, 64, 128, 256$ are shown Fig. 6a. NMF time is found to increase with *k* and to decrease with the increasing number of compute nodes. Good strong scaling is indicated by a linear decrease of NMF time with increasing compute grid size, and such behavior is only observed in select parts of the obtained results. Strong scaling is maintained up to a count of 8 nodes when $k = 8$, then to 4 nodes when $k = 16$, and lost when $k > 16$. Identical scaling is observed for cases where *A* is sparse, as shown in Fig. 6b.

The worst case scenarios, when $k = 256$, can be diagnosed from breakdown of H_{update}, W_{update} and combined all-reduce-sum (AR) execution time, as detailed in Fig. 6c. H_{update} is shown to maintain good scaling at all compute grid sizes, while W_{update} had poor scaling at each tested compute grid size. W_{update}'s poor scaling is strongly influenced by AR communications time, which already makes up more than 80% of W_{update} at 2 node count, which increases non-linearly with node count. At full grid size, AR time makes up more than 98% of W_{update}, influencing the
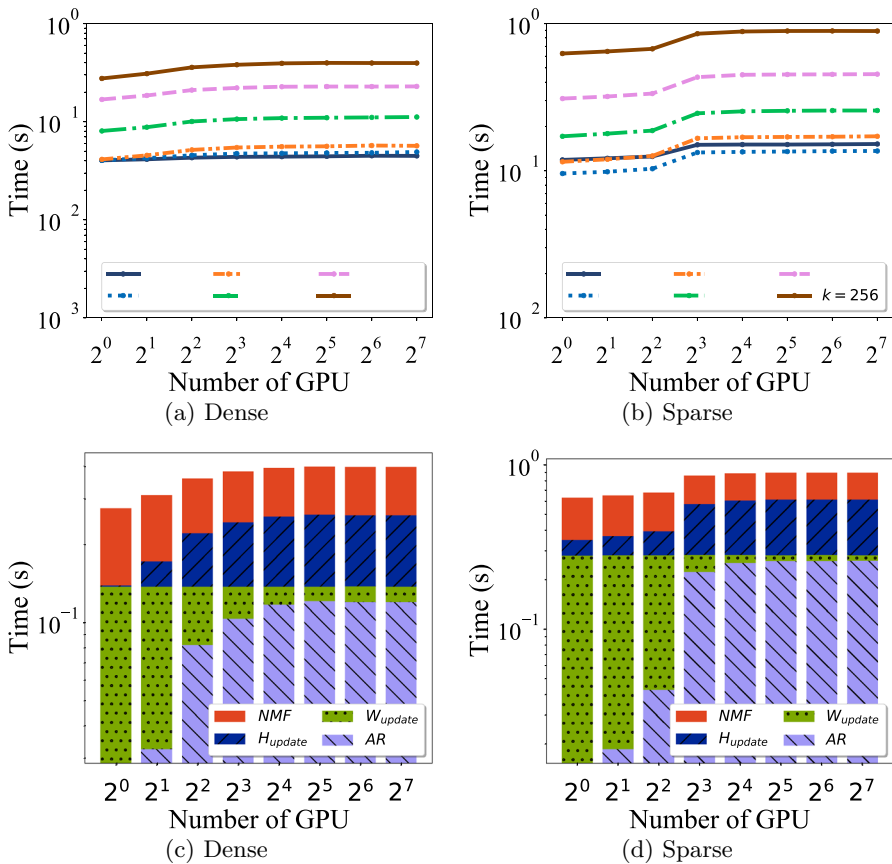
Fig. 7 Results of weak scaling study performed on *Kodiak*. NMF time vs number of GPU for various k are respectively shown in (7a) and (7b), for dense and sparse A. For the case $k = 8$, execution time of H_{update}, W_{update} and All-reduce communication are compared in (7c) and (7d), respectively for dense and sparse A

overall NMF time dominated by W_{update} time. The same explanation applies to cases where A is sparse, as one can interpret from Fig. 6d.

4.3.2 Weak scalability

Weak scaling results for cases with $k = 8, 16, 32, 64, 128, 256$ are shown Fig. 7a. Good weak scaling is indicated by constant NMF time with the increasing number of compute units, and this is observed only when $N > 8$. The lack of scaling when $N < 8$ can be explained using the breakdown of H_{update}, W_{update} and combined AR execution time for the case where $k = 256$, shown in Fig. 7c. While W_{update} maintains a perfect weak scaling at all N, H_{update} is influenced by AR communications time, which increases with GPU count. Communication grows with noticeable transitions indicating the use of slower channels. The first transition is from $N = 1$ to
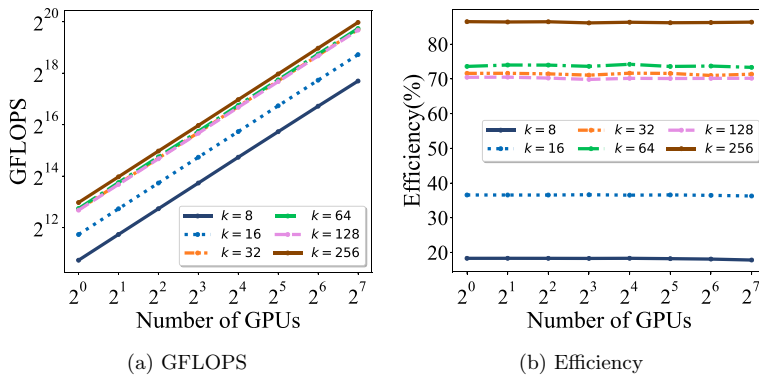
Fig. 8 FLOPS and Efficiency graph for weak scaling results for Kodiak Cluster are shown respectively in (**a**) and (**b**)

$N = 2$, indicating the beginning of *intra − node* communication between GPUs on the same node. While growing with N, *intra − node* communication remains a small portion of W_{update} ($\sim 10\%$). The next major transition occurs between $N = 4$ and $N = 8$, indicating the beginning of *inter − node* communication, which quickly saturates to $\sim 40\%$ of W_{update} by $N = 32$. Identical weak scaling is observed for cases where A is sparse, as shown by plots in Fig. 7b, and the explanation for lack of scaling when $N < 8$ is consistent with the explanation given above for the case where A is dense, as one can interpret from Fig. 7d.

In Fig. 8, we display the GFLOPS and Efficiency results generated from our weak scaling experiments conducted on the Kodiak cluster. Notably, GFLOPS shows a linear progression as GPU counts rise in Fig. 8a, indicating an efficient distribution of computational workload across GPUs. Simultaneously, the consistent relationship of Efficiency with increasing GPU counts shown in Fig. 8b underscores the effective GPU utilization, thereby confirming our implementation's efficacy in maintaining performance at scale, specially for larger ranks(k).

While all scaling results were obtained with *RNMF*, similar results will be obtained with A^T using *CNMF*.

4.4 Scaling benchmark results on *Big Data*

It's important to note that as technology continues to evolve, the scale of data storage and processing capabilities will likely increase, leading to even more significant data sets in the future. "The world's most valuable resource is no longer oil, but data1" [40]. In national security and related research efforts, vast amounts of high-dimensional data are continuously being generated by massive computer simulations, large-scale experiments, surveillance systems, etc [41, 42]. For example, Stanford Synchrotron Radiation Lightsource experiments at SLAC laboratory for revealing the inner structure of materials at nanometer scales [43, 44] and the Large Hadron Collider [45] produce terabytes of data in minutes. Another example is the petabytes of data generated by mission-critical
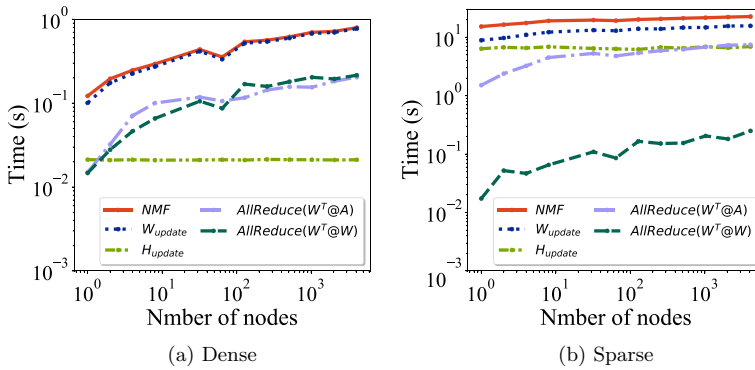
Fig. 9 Results of weak scaling study for dense and sparse *A* performed on *Summit* are shown respectively in (**a**) and (**b**)

simulations [46–50]. Exploration and analysis of such extra-large data mandates the development of novel machine learning (ML) approaches that are able to extract meaningful basic processes and fundamental features underlying the data [51].

Given our interest in exascale data, the proposed implementation was tested on a dense matrix of shape [2618523648; 32768] with a size of $\sim 340TB$, and a sparse matrix of shape $[2.89 * 10^{12}, 1.05 * 10^6]$ with sparsity 10^{-6} and size of $\sim 11EB$ ($\sim 34TB$ when compressed in a sparse format). Benchmarks were performed on *Summit*, with an allocation of 4096 nodes with 6 GPUs of 16 GB VRAM each, totaling a combined 394TB VRAM. While that is not enough to efficiently factorize either of the two matrices, we chose to cache *A* and co-factors and batch the compute of heavy, intermediate products (OOM-0). This way, we can reduce performance loss by avoiding unnecessary data transfers from host to device and vice-versa.

On the one hand, the weak scaling benchmark results for the dense array are reported in Fig. 9a. The H_{update} is shown with a perfect weak scaling, while the W_{update} is shown not to scale appropriately. Loss of scaling in the W_{update} is a consequence of the high communication cost associated with the *All-reduce* of W^{TA} and W^{TW}, which combined, make up a substantial portion of the W_{update}. The total NMF time, in turn, is significantly affected by the W_{update}, which takes about one order of magnitude more time to execute than the H_{update}. On the other, the weak scaling benchmark results for the sparse array, reported in Fig. 9b, indicate both W_{update} and H_{update} to have an excellent weak scaling. The $AR(W^{\text{TW}})$ is similar in both cases, as W^{TW} is of shape $k \times k$, but the $AR(W^{\text{TA}})$ is two orders of magnitude higher in the case of the spare dataset, proportional to *n* which is also two orders of magnitude higher. Unlike in the case of the dense array, the communication cost associated with the $AR(W^{\text{TA}})$ and $AR(W^{\text{TW}})$, although higher, are not significant enough to affect the W_{update}, consequently do not affect the overall scaling of the NMF.

(a) OOM Memory peak
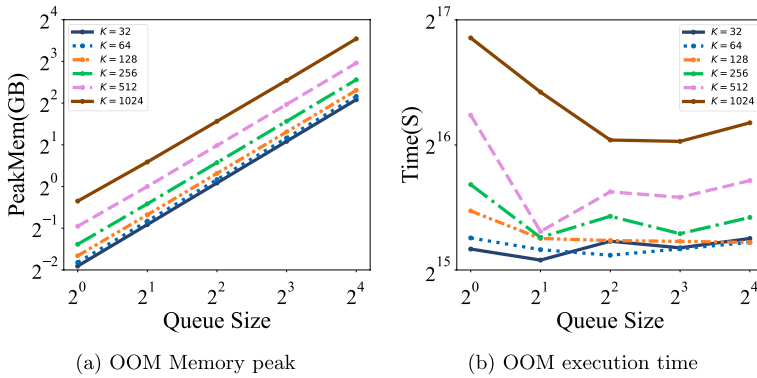
(b) OOM execution time

Fig. 10 Results of Out of memory NMF benchmarks on *Chicoma* showing **a** NMF peak memory vs queue sizes for different k, and **b** NMF execution time vs queue sizes for different k

4.5 Benchmark results on out-of-memory problems

Next, we assess the effectiveness of the proposed batching technique for OOM scenarios and the use of the CUDA stream queues to reduce communication in Algorithm 5. To this end, the proposed implementation is tested in an OOM-1 scenario, where a matrix of shape [524288, 4096] is factorized for $k = [32, 64, 128, 256, 512, 1024]$. Smaller array H is cached on GPU memory, and large arrays A and W are stored on the host and batched to GPU as needed. For this experiment, the number of iterations in Algorithm 5(line 4) fixed to *max_iters* = 100, and the number of batches is fixed to $n_b = 32$. Given the size of A in single precision is $S_A = 8$GB, the resulting batch size is $S_B = p \times n \sim 0.25$GB. The GPU peak memory utilization and NMF execution time for the 100 iteration, vs queue size, are respectively reported in Fig. 10a and Fig. 10b.

In Fig. 10a, the peak memory utilization measured when $q_s = 1$ is $S_{nmf} \sim 0.267GB$ which is close to the estimated memory complexity of $\mathcal{O}(p \times n \times q_s) \approx 0.25$GB in section 3.2, and which is a very big saving, $\sim 1/100X$, compared to the estimated $S_{NMF} \; 3 \times S_A \approx 24$ GB require by a normal implementation. This memory complexity is maintained for all k values and all queue sizes as indicated by the lines with the same slope ~ 0.267 in Fig. 10a. The increase in peak memory with increasing k for any given queue size is explained by the increase in the size of the arrays cached on GPU (H), as well as the increase in the size of the computed intermediate products (see Fig. 4. Similarly, for each k value, we note an increase in peak memory utilization with the increasing number of batches which is simply explained by the aggregated memory utilization from the concurrent streams. While from this figure, it seems unproductive to use larger stream queue sizes due to the increase in peak memory utilization, the benefits of such design choice are explained in the execution benchmark results reported in Fig. 10b.

From Fig. 10b, we first see that it is, in all cases, a good idea to choose a queue size $q_s > 1$ if one wants to speed up the *NMFk* execution time. This is explained by using large stream queue sizes makes more streams available to
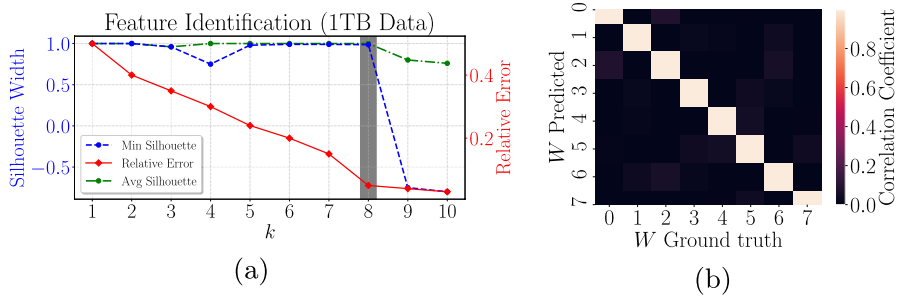
Fig. 11 **a** Estimation of number of hidden features ($k = 8$) through Silhouette analysis [6]. **b** Pearson correlation between columns of ground truth W and reconstructed W

overlap memory copies, all-reduce communications, and compute concurrently. It is, however, not the case that more streams will always make this process better, as we can see it not being the case when $q_s = 16$, where the *NMFk* execution time is not optimum for any k value. This is explained by the fact that CUDA core counts are limited and that some streams will block and wait when all cores are busy processing other streams, causing load-balancing delays. Consequently, it is crucial to fine-tune q_s for a given batch size and k to obtain optimal performance.

4.6 Validation of the model selection capability

To demonstrate the correctness of the proposed algorithm on big synthetic datasets, we first integrate our *pyDNMF-GPU* with the existing model selection algorithm *pyDNMFk* [7]. Then, we determine the number of latent features on a synthetic terabyte size matrix (with a predetermined number of features) and show that estimation is performed correctly. We generate a random matrix of dimensions 8388608×32768 as a product of two random matrices, W and H, with a latent feature count of $k = 8$. We construct W with Gaussian features with different statistical means. The *pyDNMFk-GPU* silhouette analysis corresponding to this decomposition is shown in Fig. 11a and the correctness of features is shown with confusion matrix in Fig. 11b. pyDNMFk-GPU estimates $k = 8$ as the minimum Silhouette score is high and relative error is low. For $k > 8$, the minimum silhouette score drops suddenly as the solutions begin to fit the noise Fig. 11a. Figure 11b shows a Pearson correlation matrix that illustrates a large correlation between the features of ground truth W *Ground truth* and the corresponding pyDNMFk-GPU extracted $W Predicted$ for $k = 8$. The analysis took approximately 1 h to correctly estimate the latent features on *Kodiak*. The average reconstruction error for the data is $\sim 4\%$ with the Frobenius norm objective and MU update optimization. Our experiment demonstrates that *pyDNMFk-GPU* correctly estimates the number of latent features in addition to its scalability for large datasets as demonstrated in previous sections.

5 Conclusion

In summary, we demonstrated a novel scalable and portable framework, *pyDN-MFk-GPU*, for non-negative matrix factorization based on custom multiplicative updates, with automatic determination of the number of latent features on Exascale data. Scalability of the framework was demonstrated via strong and weak scaling benchmarks, and speedup gains on GPU over CPU were found to vary with *k* and to increase with the size of the HPC system. The efficacy of the proposed tiling technique was demonstrated through the OOM-0 problem by factorizing a dense dataset of 340TB and a sparse dataset of size 11EB, where the implementation was found to have good week scaling on upto to 25k GPU. We also demonstrated the efficacity of the proposed batching technique along with the importance of using CUDA streams by solving *OOM-1* problem, where memory complexity was shown to be of the $\mathcal{O}(p \times n \times q_s)$, resulting in a significant saving of $\sim 100X$ smaller peak memory utilization in some cases. The automatic model selection capability was verified by correctly decomposing large synthetic data with a predetermined number of latent features and factors.

Data availability The code and the benchmark results used in this paper will be available at https://github.com/lanl/pyDNMFk.

Declarations

Conflict of interest The authors declare that they have no competing interest.

References

1. Lee DD, Seung HS (1999) Learning the parts of objects by non-negative matrix factorization. Nature 401(6755):788–791
2. Cichocki A, Zdunek R, Phan AH, Amari S-i (2009) Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation
3. Everett B (2013) An introduction to latent variable models
4. Alexandrov LB, Nik-Zainal S, Wedge DC, Campbell PJ, Stratton MR (2013) Deciphering signatures of mutational processes operative in human cancer. Cell Rep 3(1):246–259
5. Alexandrov BS, Alexandrov LB, Iliev F, Stanev VG, Vesselinov V (2020) Source identification by non-negative matrix factorization combined with semi-supervised clustering. Google Patents. US Patent 10,776,718
6. Chennupati G, Vangara R, Skau E, Djidjev H, Alexandrov B (2020) Distributed non-negative matrix factorization with determination of the number of latent features. The Journal of Supercomputing, 1–31
7. Bhattarai M, Nebgen B, Skau E, Eren M, Chennupati G, Vangara R, Djidjev H, Patchett J, Ahrens J, ALexandrov B (2021) pyDNMFk: python distributed non negative matrix factorization. GitHub. https://doi.org/10.5281/zenodo.4722448
8. Vangara R, Bhattarai M, Skau E, Chennupati G, Djidjev H, Tierney T et al (2021) Finding the number of latent topics with semantic non-negative matrix factorization. IEEE Access, pp 117217–117231
9. Alexandrov LB, Nik-Zainal S, Wedge DC, Aparicio SA, Behjati S, Biankin AV, Bignell GR, Bolli N, Borg A, Børresen-Dale A-L et al (2013) Signatures of mutational processes in human cancer. Nature 500(7463):415
10. Alexandrov LB, Kim J, Haradhvala NJ, Huang MN, Ng AWT, Wu Y, Boot A, Covington KR, Gordenin DA, Bergstrom EN et al (2020) The repertoire of mutational signatures in human cancer. Nature 578(7793):94–101
11. Vangara R, Skau E, Chennupati G, Djidjev H, Tierney T, Smith JP, Bhattarai M, Stanev VG, Alexandrov BS (2020) Semantic nonnegative matrix factorization with automatic model determination for topic modeling, pp 328–335. IEEE
12. Bhattarai M, Chennupati G, Skau E, Vangara R, Djidjev H, Alexandrov BS (2020)Distributed nonnegative tensor train decomposition. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–10. IEEE
13. Alexandrov BS, Stanev VG, Vesselinov VV, Rasmussen KØ (2019) Nonnegative tensor decomposition with custom clustering for microphase separation of block copolymers. Stat Anal Data Min ASA Data Sci J 12(4):302–310
14. Pulido J, Patchett J, Bhattarai M, Alexandrov B, Ahrens J (2021) Selection of optimal salient time steps by non-negative tucker tensor decomposition. In: Agus M, Garth C, Kerren A (eds) EuroVis 2021—short papers. The Eurographics Association. https://doi.org/10.2312/evs.20211055
15. Bhattarai M, Kharat N, Skau E, Nebgen B, Djidjev H, Rajopadhye S, Smith JP, Alexandrov B (2022) Distributed non-negative rescal with automatic model selection for exascale data. arXiv preprint arXiv:2202.09512
16. Bhattarai M, Kharat N, Skau E, Truong D, Eren M, Rajopadhye S, Djidjev H, Alexandrov B pyDRESCALk: python distributed non negative RESCAL decomposition with determination of latent features. https://doi.org/10.5281/zenodo.5758446
17. Eren ME, Moore JS, Skau E, Moore E, Bhattarai M, Chennupati G, Alexandrov BS (2022) General-purpose unsupervised cyber anomaly detection via non-negative tensor factorization. Research and practice, digital threats
18. Eren ME, Richards LE, Bhattarai M, Yus R, Nicholas C, Alexandrov BS (2022) Fedsplit: One-shot federated recommendation system based on non-negative joint matrix factorization and knowledge distillation. arXiv preprint arXiv:2205.02359
19. Eren ME, Solovyev N, Bhattarai M, Rasmussen K, Nicholas C, Alexandrov BS (2022) Senmfk-split: Large corpora topic modeling by semantic non-negative matrix factorization with automatic model selection. arXiv preprint arXiv:2208.09942
20. Févotte C, Cemgil AT (2009) Nonnegative matrix factorizations as probabilistic inference in composite models. In: 2009 17th European Signal Processing Conference, pp 1913–1917. IEEE

21. Phan AH, Cichocki A (2008) Multi-way nonnegative tensor factorization using fast hierarchical alternating least squares algorithm (HALS). In: Proc. of The 2008 international symposium on nonlinear theory and its applications
22. Kim J, Park H (2012) Fast nonnegative tensor factorization with an active-set-like method, pp 311–326
23. Kim J, He Y, Park H (2014) Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework. J Global Optim 58(2):285–319
24. Battenberg E, Wessel D (2009) Accelerating non-negative matrix factorization for audio source separation on multi-core and many-core architectures. In: ISMIR, pp 501–506
25. Fairbanks JP, Kannan R, Park H, Bader DA (2015) Behavioral clusters in dynamic graphs. Parallel Comput 47:38–50
26. Moon GE, Ellis JA, Sukumaran-Rajam A, Parthasarathy S, Sadayappan P (2020) ALO-NMF: Accelerated locality-optimized non-negative matrix factorization. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 1758–1767
27. Phipps ET, Kolda TG (2019) Software for sparse tensor decomposition on emerging computing architectures. SIAM J Sci Comput 41(3):269–290
28. Mejía-Roa E, Tabas-Madrid D, Setoain J, García C, Tirado F, Pascual-Montano A (2015) NMF-mGPU: non-negative matrix factorization on multi-GPU systems. BMC Bioinf 16(1):1–12
29. Lopes N, Ribeiro B (2010) Non-negative matrix factorization implementation using graphic processing units. In: International Conference on Intelligent Data Engineering and Automated Learning, pp 275–283. Springer
30. Kannan R, Ballard G, Park H (2016) A high-performance parallel algorithm for nonnegative matrix factorization. ACM SIGPLAN Not 51(8):1–11
31. Koitka S, Friedrich CM (2016) nmfgpu4R: GPU-Accelerated Computation of the Non-Negative Matrix Factorization (NMF) Using CUDA Capable Hardware. R J 8(2):382
32. Tang B, Kang L, Zhang L, Guo F, He H (2021) collaborative filtering recommendation using nonnegative matrix factorization in GPU-accelerated spark platform. Scientific Programming **2021**
33. Eswar S, Hayashi K, Ballard G, Kannan R, Matheson MA, Park H (2021) PLANC: parallel low-rank approximation with nonnegativity constraints. ACM Trans Math Softw 47(3):1–37
34. Boureima I, Bhattarai M, Eren ME, Solovyev N, Djidjev H, Alexandrov BS (2022) Distributed out-of-memory SVD on CPU/GPU architectures. arXiv preprint arXiv:2208.08410
35. Okuta R, Unno Y, Nishino D, Hido S, Loomis C (2017) Cupy: A NumPy-compatible library for NVIDIA GPU calculations. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS). http://learningsys.org/nips17/assets/papers/paper_16.pdf
36. ...Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, Kern R, Picus M, Hoyer S, van Kerkwijk MH, Brett M, Haldane A, del Río JF, Wiebe M, Peterson P, Gérard-Marchant P, Sheppard K, Reddy T, Weckesser W, Abbasi H, Gohlke C, Oliphant TE (2020) Array programming with NumPy. Nature 585(7825):357–362. https://doi.org/10.1038/s41586-020-2649-2
37. Dalcin L, Fang Y-LL (2021) mpi4py: Status update after 12 years of development. Comput Sci Eng 23(4):47–54
38. ...Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, Polat İ, Feng Y, Moore EW, VanderPlas J, Laxalde D, Perktold J, Cimrman R, Henriksen I, Quintero EA, Harris CR, Archibald AM, Ribeiro AH, Pedregosa F, van Mulbregt P (2020) SciPy 1.0 contributors: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat Methods 17:261–272. https://doi.org/10.1038/s41592-019-0686-2
39. Awan AA, Hamidouche K, Venkatesh A, Panda DK (2016) Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In: Proceedings of the 23rd European MPI Users' Group Meeting, pp 15–22
40. Quigley E, Holme I, Doyle DM, Ho AK, Ambrose E, Kirkwood K, Doyle G (2021) data is the new oil: citizen science and informed consent in an era of researchers handling of an economically valuable resource. Life Sci Soc Policy 17(1):1–13
41. Hickey A (2019) Zettabytes of data hog up space and resources
42. Akhgar B, Saathoff GB, Arabnia HR, Hill R, Staniforth A, Bayerl PS (2015) Application of Big Data for national security: a practitioner's guide to emerging technologies. Butterworth-Heinemann, Oxford

43. Sierra RG, Laksmono H, Kern J, Tran R, Hattne J, Alonso-Mori R, Lassalle-Kaiser B, Glöckner C, Hellmich J, Schafer DW et al (2012) Nanoflow electrospinning serial femtosecond crystallography. Acta Crystallogr D Biol Crystallogr 68(11):1584–1587
44. Sandberg RL, Huang Z, Xu R, Rodriguez JA, Miao J (2013) Studies of materials at the nanometer scale using coherent x-ray diffraction imaging. JOM 65:1208–1220
45. Butter A, Plehn T, Schumann S, Badger S, Caron S, Cranmer K, Di Bello FA, Dreyer E, Forte S, Ganguly S et al (2023) Machine learning and LHC event generation. SciPost Phys 14(4):079
46. Gubaev K, Podryabinkin EV, Shapeev AV (2018) Machine learning of molecular properties: locality and active learning. J Cheml Phys 148(24):241727
47. Kruglov I, Sergeev O, Yanilkin A, Oganov AR (2017) Energy-free machine learning force field for aluminum. Sci Rep 7(1):8512
48. Haghighatlari M, Heidar-Zadeh F, Hirn M, Hoja J, Isayev O, Kondor R, Li L, Li Y, Martyna G, Meila M et al (2017) IPAM program on machine learning & many-particle systems-recent progress and open problems
49. Messina P, Lee S (2016) The us exascale computing project. In: Proc. ACM/IEEE conf. supercomputing (birds a feather)
50. Zhang J, Xiao M, Gao L (2019) An active learning reliability method combining kriging constructed with exploration and exploitation of failure region and subset simulation. Reliab Eng Syst Saf 188:90–102
51. Franke B, Plante J-F, Roscher R, E-sA Lee, Smyth C, Hatefi A, Chen F, Gil E, Schwing A, Selvitella A et al (2016) Statistical inference, learning and models in big data. Int Stat Rev 84(3):371–389

Authors and Affiliations

Ismael Boureima[1] · Manish Bhattarai[1] · Maksim Eren[1] · Erik Skau[2] · Philip Romero[3] · Stephan Eidenbenz[2] · Boian Alexandrov[1]

✉ Ismael Boureima
iboureima@lanl.gov

Manish Bhattarai
ceodspspectrum@lanl.gov

Maksim Eren
maksim@lanl.gov

Erik Skau
ewskau@lanl.gov

Philip Romero
prr@lanl.gov

Stephan Eidenbenz
eidenben@lanl.gov

Boian Alexandrov
boian@lanl.gov

[1] Theoritical Divison, Los Alamos National Laboratory, Los Alamos 87545, NM, USA

[2] Computer, Computational, and Statistical Science Division, Los Alamos National Laboratory, Los Alamos 87545, NM, USA

[3] HPC Divison, Los Alamos National Laboratory, Los Alamos 87545, NM, USA