Distributed Out-of-Memory SVD on CPU/GPU Architectures

Ismael Boureima  
*Theoretical Division*  
LANL  
Los Alamos, U.S  
iiboureima@lanl.gov

Manish Bhattarai  
*Theoretical Division*  
LANL  
Los Alamos, U.S  
ceodspspectrum@lanl.gov

Maksim E. Eren  
*Theoretical Division*  
LANL  
Los Alamos, U.S  
maksim@lanl.gov

Nick Solovyev  
*Theoretical Division*  
LANL  
Los Alamos, U.S  
nks@lanl.gov

Hristo Djidjev  
*Information Systems*  
LANL Los Alamos, U.S  
djidjev@lanl.gov

Boian S. Alexandrov  
*Theoretical Division*  
LANL  
Los Alamos, U.S  
boian@lanl.gov

Abstract—We propose an efficient, distributed, out-of-memory implementation of the truncated singular value decomposition (t-SVD) for heterogeneous high performance computing (HPC) systems. Various implementations of SVD have been proposed, with most only estimate the singular values as the estimation of the singular vectors can significantly increase the time and memory complexity of the algorithm. In this work, we propose an implementation of SVD based on the power method, which is a truncated singular values and singular vectors estimation method. Memory utilization bottlenecks in the power method used to decompose a matrix A are typically associated with the computation of the Gram matrix A^T A, which can be significant when A is large and dense, or when A is super-large and sparse. The proposed implementation is optimized for out-of-memory problems where the memory required to factorize a given matrix is greater than the available GPU memory. We reduce the memory complexity of A^T A by using a batching strategy where the intermediate factors are computed block by block, and we hide I/O latency associated with both host-to-device (H2D) and device-to-host (D2H) batch copies by overlapping each batch copy with compute using CUDA streams. Furthermore, we use optimized NCCL based communicators to reduce the latency associated with collective communications (both intra-node and inter-node). In addition, sparse and dense matrix multiplications are significantly accelerated with GPU cores (or tensors cores when available), resulting in an implementation with good scaling. We demonstrate the scalability of our distributed out-of-core SVD algorithm to successfully decompose dense matrix of size 1TB and sparse matrix of 1e-6 sparsity with size of 128 PB in dense format.

Index Terms—SVD, out-of-memory, latent features, data compression, distributed processing, parallel programming, big data, heterogeneous HPC systems, GPU, CUDA, NCCL, cupy

I. INTRODUCTION

Singular value decomposition (SVD) decomposes a matrix A of size m×n into two orthogonal matrices U of size m×m, V of size n×n, and a diagonal matrix Σ of size m×n, which is a non-negative matrix, such that A = UΣVT. The diagonal elements of Σ are also called singular values, whereas U and V contain left and right singular vectors of A, respectively.

The power method is one way of estimating the SVD. The power method computes the first k monotonically decreasing singular values and the corresponding vectors, saving the computational need to estimate the complete singular values and vectors [2]. The power method based SVD is also known as truncated SVD [14], which is given as $A = U_k \Sigma_k V_k^T$. Here $\Sigma_k$ is a diagonal matrix comprising singular values $\sigma_1 > \sigma_2 > \ldots > \sigma_k$, and $U_k$ and $V_k$ are the orthogonal matrices of sizes $m \times k$ and $k \times n$, respectively, whose columns are the left and right singular vectors corresponding to the first k largest singular values. The ability to estimate the first k large singular values and corresponding singular vectors addresses the challenge in estimating all the singular values and vectors. The pseudocode of the truncated SVD is shown in Algorithms 1 and 2.

### Algorithm 1 SVD(A, $\epsilon$, k = -1) - Truncated SVD

Require: $A \in \mathbb{R}^{m \times n}$ and a scalar $\epsilon$ where k is an optional parameter.

1. $m, n = \text{shape}(A)$
2. if k == -1 then
3. $k = \min(m, n)$
4. $U, V, \sigma = [[||u||]], [||v||]$  
   \hspace{1cm} $\triangleright$ Initialize as empty arrays
5. for $l$ in [1, k] do
6. $X = A$
7. if $l > 1$ then
8. $X = X - U(:, l) \text{diag}(\sigma[l]) V(:, l)^T$
9. end
10. if $m > n$ then
11. $U(l) = \text{SVDD1D}(X, \epsilon)$
12. $U(l) = A U(l)$
13. $\sigma(l) = ||V(l)||$
14. $U(l) = U(l) / \sigma(l)$
15. else
16. $U(l) = \text{SVDD2D}(X, \epsilon)$
17. $V(l) = A^T \oplus U(l)$
18. $\sigma(l) = ||V(l)||$
19. $V(l) = V(l) / \sigma(l)$

Ensure: $U \in \mathbb{R}^{m \times k}, \sigma \in \mathbb{R}^k, V \in \mathbb{R}^{n \times k}$

Ensure: $UU^T = I_m \times m, VV^T = I_n \times n$, where $I$ is Identity matrix and diag($\Sigma$) = $\sigma$ where $\sigma = (\sigma_1, \sigma_2, \ldots \sigma_k)$ and $\Sigma \in \mathbb{R}^{k \times k}$

SVD provides an advantage in decomposing large-scale
Algorithm 2 SVD_1D(X, ε)

Require: X ∈ R^m×n and a scalar ε.
1: m, n = shape(X)
2: k = min(m, n)
3: x ≈ N(0, 1) where x, 0, 1 ∈ R^k
4: $X \approx \Sigma V T$, assuming $k < < (m, n)$. Starting counting at line 6
5: set $v_0 = x$
6: if $m > n$ then
7:  $B = X^T @ X$
8: else
9:  $B = X @ X^T$
10: while true do
11:  $v_1 = B @ v_0$
12:  $v_0 = x - \frac{v_1}{|v_1|}$
13:  if $|v_0| / |v_1| \geq 1 - \epsilon$ then
14:  return $v_1$
15:  $v_0 = v_1$

datasets into low-dimensional factors, thus allowing compressed representations. As computing power has increased with the introduction of modern GPUs and TPUs, SVD (along with many other ML frameworks) has seen significant acceleration on this hardware. However, the ever-growing volume of data produced by social networks, medical applications, and experimental simulations has led to SVD decomposition bottlenecks in single computing hardware as the data does not fit in memory. In addition to these memory constraints, decomposition of such large-scale datasets requires significant computational resources along with data storage and movement. Almost every work in the relevant literature aims to solve only one or two of these challenges where the results do not show good scalability. This is due to the significant communication cost associated with data movement across different computing elements, which often exceeds the computation cost of these algorithms.

This mandates the need for distributed algorithms with the additional ability for out-of-memory computation that would enable performing computation over distributed hardware efficiently without memory bottleneck while also computing the singular vectors. In such a design, the data blocks that are not used in computation reside on the external disk or outside the memory of each node’s modern GPU/TPU hardware. Furthermore, to make an efficient design, there is a need to overlap the data IO with computation operation to minimize the overall latency in performing decomposition. In this work, we devise a novel parallel SVD framework called pyDSVD-GPU, which combines batching for out-of-memory and tiling for distributed computation of large-sized sparse/dense datasets while reducing the communication and data movement cost on heterogeneous hardware. The main contributions of this paper include:

- The first novel distributed algorithm with out-of-memory support for SVD for sparse and dense matrices on CPU/GPU hardware.
- The first NCCL Communicator accelerated SVD decomposition tool in distributed GPUs.
- Custom SVD algorithm for decomposition of very large dense and sparse datasets.

II. RELATED WORK

SVD is an integral component of many machine learning algorithms involving dimensionality reduction, data compression [3], knowledge graphs [4], and information retrieval [5]. With the growing volume of data, it is becoming impractical for some of these machine learning systems, such as social network analytics, recommendation systems, autonomous driving, and data compression, to name a few, to operate. This problem has motivated the need for out-of-memory implementations of SVD for distributed systems. Various out-of-memory algorithms and distributed algorithms have been recently devised for either CPU or GPU hardware. [11] provides an overview of different large-scale SVD frameworks designed to decompose large-scale datasets. The work in papers [8], [10], [16], [20] demonstrate the out-of-memory SVD implementation for CPU hardware whereas papers [6], [12], [13], [18] demonstrate the implementation for the GPU hardware. A majority of the existing out-of-memory (OOM) and distributed implementation of SVD emphasize computing only the singular values, which is insufficient for applications such as tensor networks, which need accurate computation of a substantial number of singular vectors. In addition, a majority of them can only decompose a matrix of size $m \times n$ when $m \gg n$ or $m \ll n$ as the data partitioning is done along one axis. However, most real-world datasets such as knowledge graphs tend to have symmetrical matrix shapes, i.e., $m \approx n$, and one can not apply such distributed/OOM implementations. Furthermore, most of the OOM and the distributed implementations exist in the CPU. Still, the data size becomes a bottleneck given the hardware constraint for these implementations. Also, the distributed implementations for GPU discussed in [17], [22] do not report the scalability for very large datasets. Hence it is vital to take advantage of both the out-of-memory ability combined with the distributed implementation to decompose matrices of unprecedented sizes at an accelerated pace on GPU. This approach has recently been utilized in our distributed out-of-memory non-negative matrix factorization (NMF) paper [7] for type 0 out-of-memory situations (OOM-0)(discussed later). In such distributed out-of-memory implementation, the distributed axis splits one of the co-factor matrices. In contrast, the batched axis splits the other co-factor matrix enabling the decomposition of massive sparse datasets. Furthermore, our distributed implementation is optimized with NCCL [11] communicator for accelerated inter and intra GPU communication. In contrast, CUDA streams accelerate the data movement between GPU and CPU for out-of-memory implementation.

III. RATIONALE FOR AN ALGORITHM FOR THE OUT-OF-MEMORY DISTRIBUTED SVD

The proposed implementation of SVD is for heterogeneous HPC systems, with the ability to handle OOM scenarios, where the data is too big to be cached on combined GPU memory. A serial algorithm for the truncated SVD is given in Algorithm 1, from which we can estimate the memory complexity of SVD for a given matrix $A$ of size $S_A$ (in bytes) to be $S_{SV D} \approx 4 \times S_A$, assuming $k << (m, n)$. Starting counting at line 6
of Algorithm 1, two buffers of size $S_A$ are required for the copy $X = A \sim 2 \times S_A$ total, the memory utilization will peak at line 8 of Algorithm 1, where two additional buffers of size $S_A$ will be required to compute the residual $X = X - U[:: l] diag(\sigma[:: l]) V[:: l]^T$ (totalling $4 \times S_A$). Note that when $A$ is sparse, choosing to represent $A$ in a sparse format, such as Compressed Sparse Row (CSR) or Coordinate Format (COO), can dramatically reduce the memory required to store $A$. This memory reduction can skew the calculation of $S_{SV D}$ because the resulting co-factors $U$ and $V$ are dense, and most importantly, the residual $X$ is now dense with the same shape as $A$, and now $S_{SV D} \approx 2 \times S_A$, where $S_A$ is the size of the dense representation of $A$.

When performing SVD on GPUs, OOM situations can arise in various scenarios with different degrees of complexity. 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 $U$, $\Sigma$, and $V$ can easily be stored on GPU memory, but an explosion of memory requirement occurs when computing intermediate results. This is often the case when computing the intermediate results $X = X - U \Sigma V^T$ (line 8 of Algorithm 1), when $A$ is a large sparse matrix of very low density and the factors are dense. The matrix $X$ 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 $S_A \sim 3 \times S_A \times 12 \approx 12 GB$ (the factor of 3 accounts for storing the data, indices and index pointers for CSR format), consequently $S_{SV D} \approx 2 \times S_A \approx 4TB$. Assuming very small $k$, $A$ and all co-factors can be stored on GPU, however the calculation of the residual $X$ from $X = X - U \Sigma V^T$ would still require a whopping $\sim 8000GB$ of GPU memory (line 9 of Algorithm 1), making this scenario a OOM-0 problem.

Below we propose a solution to address this bottleneck, where we avoid the direct matrix-matrix multiplications that would result in such a large dense matrix. Instead, we perform a series of matrix-vector computations to minimize the overall computation and memory cost.

A more complex OOM scenario, type 1 (OOM-1), arises in cases where matrix $A$ and at most two 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. We propose tiling and batching techniques to deal with OOM-0 and OOM-1 problems. Both tiling and batching are block-based computation techniques that operate the same way; however, in the interest of clarity, we differentiate the two base on the employed data transfer pipelines. Tiling takes place on GPU and employs GPU data transfer pipelines such as global memory to shared memory data transfer pipelines, while batching takes place between host (CPU) and device (GPU) and uses the host to device data transfer pipelines and vice-versa. Optimization of tiling techniques relies on GPU architecture and features such as memory speed, available VRAM, etc. In contrast, the optimization of batching techniques depends more on the speed of busses connecting host and device, such as PCIe or NV-Link bus speeds, as well as the number of CUDA streams, which can be exploited in an asynchronous approach to overlap data transfer and compute operation to hide data copy latency. We adopt tiling techniques to deal with OOM-0 problems, and batching technique to han-
dle OOM-1 problems. In extreme cases, we will complement batching by tiling to further reduce memory footprint. Bellow, we discuss our implementation and design choices.

IV. OUT OF MEMORY ALGORITHM DESIGN FOR LARGE SPARSE DATASETS

The power method approach to performing truncated SVD is a popular approach that is based on repeatedly performing a linear projection on an initial vector until it converges to the direction of each desired eigenvector. A serial version of the truncated SVD algorithm is given in Algorithm 1, which relies on Algorithm 2 for estimating $k$ singular vectors where each singular vector is estimated one at a time. The algorithm’s major computational/memory bottlenecks are in the computation of the Gram matrix (i.e., lines 7 and 9) in Algorithm 2. In addition, the computation of line 9 in Algorithm 1 requires the computation of the product of the SVD factors requiring additional memory. The distributed realiztion of the Gram matrix computation is presented in Algorithm 3. We utilize this distributed gram-based realization of Algorithm 2 for dense datasets. However, due to the bottleneck associated with computing the gram of residual matrix from very large dense cofactors for the sparse dataset, we avoid this distributed realization for the sparse dataset and propose a new realization discussed next. We reduce memory complexity using the analytical derivation below:

From the line 8 of Algorithm 1, we have the following expression:

$$X' = X - U[:, l] \text{diag}(\sigma[l])V[:, l]^T$$

let $U = U[:, l]$, $V = V[:, l]$ and $\Sigma = \text{diag}(\sigma[l])$. Then for $m > n$, we have

$$B = X'^T X' = (X - U \Sigma V^T)^T(X - U \Sigma V^T)$$

$$= (X^T - V \Sigma^T U^T)(X - U \Sigma V^T)$$

$$= X^T X - V \Sigma^T U^T X - X^T U \Sigma V^T + V \Sigma^T U^T U \Sigma V^T$$

$$= X^T X - V \Sigma^T U^T X - X^T U \Sigma V^T + V \Sigma^2 V^T,$$

(1)

since $U^T U = I$ and $\Sigma \Sigma^T = \Sigma^2$ as $\Sigma$ is a diagonal matrix. Still, Equation 1 is computationally and memory expensive due to matrix-matrix multiplications and requirement to store $n \times n$ dense matrix. So we avoid the direct computation of Equation 1, but instead, we multiply the expression with vector $v_0$ as shown in line 11 of Algorithm 2. So we can represent

$$v_1 = B v_0$$

$$= (X^T X - V \Sigma^T U^T X - X^T U \Sigma V^T + V \Sigma^2 V^T)v_0$$

$$= X^T v_0 - V \Sigma^T U^T X v_0 - X^T U \Sigma V^T v_0 +$$

$$V \Sigma^2 V^T v_0,$$

(2)

where each product is computed in the right-to-left order. Similarly, for $m < n$, we have:

$$v_1 = X X^T v_0 - U \Sigma V^T X v_0 - X V \Sigma V^T U^T v_0 +$$

$$U \Sigma^2 U^T v_0.$$

(3)

Equations 2 and 3 mean that we significantly reduce memory complexity by avoiding the direct calculation of the residual $X$ as well as the calculation of the Gram $B = X^T X$, and instead computing directly the $k^{th}$ singular vector skipping lines 6-9 in Algorithm 2. Evaluating the expression in Equations 2 or 3 right to the left would replace the matrix-matrix multiplications with a series of matrix-vector multiplications, thus significantly lowering memory complexities. The newly devised algorithm’s distributed implementation is presented in Algorithm 4. This can, however, increase the time complexity of the OOM-1 problem due to the need to batch in each iteration any of the matrices that are not cached on GPU. While this data movement cost is reasonable for a sparse dataset, the cost can be a significant bottleneck for dense datasets due to the considerable data movement for OOM implementation. Consequently, a design trade-off between time and memory complexity needs to be assessed based on the available hardware, i.e., this can be a viable option with modern GPUs with high-bandwidth SXM/NVLink interfaces.

V. PYDSVD FOR DISTRIBUTED HETEROGENEOUS SYSTEMS

A. Implementation of SVD for distributed heterogeneous systems

The proposed implementation of SVD for distributed heterogeneous systems partitions large SVD problems into smaller distributed problems using strong considerations for data locality, such as avoiding inefficiencies associated with communication (data transfer) in the distributed system. Additional trade-offs are considered in OOM scenarios; for instance, to reduce communication, it is sometimes better to replicate data over the distributed grid, while other times, it is acceptable to use batching/tiling techniques that can increase communication latency to lower the memory footprint. Bellow, we discuss the problem partition strategies in subsection(V-B), followed...
Algorithm 4 dist_Compute_v0(X, U, Σ, V, n_b, N)

Require: X ∈ R^{m×n}, U ∈ R^{m×k}, V ∈ R^{m×k} and Σ ∈ R^{k×k}

Require: X distributed across N GPUs where X_{ij} ∈ R^{j×k}, if m > n otherwise

1. if m >= n then
   2. I = m/N and n = n/n_b
   3. (X_{ij}) = X_{ij} ⊲ (v_{ij}) ⊢ Multiply a batch of X of size I × b with a
      batch of v_{ij} of size b × 1 so that a batch of X_{ij} is of size I for n_b batches.
   4. (X_{ij}) = Σ_{j=1}^{n_b} (X_{ij}) ⊢ Reduction along batches so that (X_{ij}) is of size
      I × 1
   5. (X_{ij}) = (X_{ij}) ⊲ (X_{ij}) ⊢ (X_{ij}) is of size b × 1 for n_b
      batches for N GPUs.
   6. (Σ_{i}) = Σ_{i} ⊲ (X_{ij}) ⊢ (X_{ij}) is of size b × 1 for n_b
      batches.
   7. X_{ij} = U_{ij} ⊲ (X_{ij}) ⊢ (X_{ij}) is of size b × 1 for n_b
      batches.
   8. V_{ij} = U_{ij} ⊲ (X_{ij}) ⊢ All reduce sum along the GPUs i.e.
      b axis such that (X_{ij}) is of size b × 1 for n_b batches along each GPU.
   9. (X_{ij}) = Σ_{j=1}^{n_b} (X_{ij}) ⊢ All reduce sum along the GPUs so that
      U_{ij} X_{ij} is of size k × 1 and same for all GPUs.
   10. (Σ_{i}) = Σ_{i} ⊲ (X_{ij}) ⊢ Σ_{i} is of size b × 1 for n_b
      batches.
   11. (V_{ij}) = (V_{ij}) ⊲ (X_{ij}) ⊢ Multiplying a batch of V of size k × b with a
      batch of V_{ij} of size I × 1 so that a batch of V_{ij} is of size k for n_b
      batches.
   12. V_{ij} = Σ_{i} ⊲ (X_{ij}) ⊢ (V_{ij}) is of size b × 1 for n_b
      batches, which is the same for every GPU.
   13. Σ_{i} = Σ_{i} ⊲ (X_{ij}) ⊢ Σ_{i} is of size k × 1
   14. (U_{ij} V_{ij}) = U_{ij} ⊲ (V_{ij}) ⊢ (V_{ij}) is of size b × 1 for n_b
      batches.
   15. (X_{ij} U_{ij} V_{ij}) = Σ_{i} ⊲ (U_{ij} V_{ij}) ⊢ (X_{ij} U_{ij} V_{ij}) is of
      size b × 1 for n_b batches.
   16. (X_{ij} U_{ij} V_{ij}) = Σ_{i} ⊲ (X_{ij} U_{ij} V_{ij}) ⊢ (X_{ij} U_{ij} V_{ij}) is of
      size b × 1 for n_b batches.
   17. (V_{ij}) = (V_{ij}) ⊲ (X_{ij}) ⊢ (V_{ij}) is of size b × 1 for n_b
      batches.
   18. (V_{ij}) = (V_{ij}) ⊲ (X_{ij}) ⊢ (V_{ij}) is of size b × 1 for n_b
      batches.
   19. v_{ij} = Σ_{i} ⊲ (X_{ij}) ⊢ (V_{ij} ⊲ (X_{ij})) ⊢ (V_{ij} ⊲ (X_{ij})) is of size
      b × 1 for n_b batches.
   20. return v_{ij}

by a discussion of our tiling and batching approaches in subsection(V-C).

B. Partition of computational work on the distributed HPC system

Our implementation considers two one-dimensional data partition strategies based on the shape of matrix A (m × n).
A column (vertical) partition, CSVD employed when n > m, and a row (horizontal) partition, HSVD, used otherwise.

Assuming a distributed system with N GPUs, where each GPU is indexed by its global rank g i.D. In the RSVD approach illustrated in Figure 1a, the i th GPU with g i.D = i will work on array partitions A[i:0, i−1:] U[i:0, i−1:] and V[i:0, i−1:]. A, U, Σ, and V, where i is i × I, i = (i + 1) × I, and I = m/N (partition size). Each GPU gets a full copy of Σ and V (Σ and V are replicated) and a unique partition of A and U. This translates into a segmentation of arrays A and U on global memory, as we illustrate for the case N = 5 in Figure 1a. The colored zones indicate local partitions of A on each GPU in global memory (data locality), and consequently help conceptualize communication requirements whenever information is exchanged from one zone to another.

C. Out-of-memory implementation

In out-of-memory situations where the available GPU memory S_G is insufficient (S_G < S_{CPU}), smaller arrays are cached on GPU memory, and larger arrays are kept on host memory. We further split the local blocks of data into smaller batches/tiles in a way that is either collinear to the direction of the larger dimension A, i.e., m in RSVD where m > n, or in a way that is orthogonal to the direction of m. In the former method, we talk about adopting a collinear batching/tiling technique; in the latter, we talk about adopting an orthogonal batching/tiling technique. To illustrate, let b_s be a batch size control parameter, and let’s assume we have a RSVD (CSVD) partition scenario. The number batches in an orthogonal batching technique is then given by n_B = n/b_s (n_B = m/b_s), and U[i, :] (V[:, J]) is cached on GPU memory whereas, heavier arrays A[i, b_0 : b_1] (A[b_0 : b_1, J]) and V[:, b_0 : b_1] (U[b_0 : b_1, J]) are batched to their respective GPUs as needed, such that for the b th batch, b_0 = b × b_s and b_1 = (b + 1) × b_s. Cartoons in Figure 1b illustrate the orthogonal tiling strategy of a RSVD partition of A among 5 GPUs. Data locality is indicated by colored zones, coded with the legend at the bottom. V and Σ are replicated on all GPUs, as indicated by green color. Vertical solid lines illustrate the segmentation of local data into batches/tiles, as is the case for A and V. Note that these scenarios assume a OOM-0 scenario because all data blocks are on GPU memory. In OOM-1, A and V would have been on host RAM memory.
Figure 2 and Algorithm 3 highlights the important aspects of computing the Gram product \( \mathbf{B} = \mathbf{A}^T \mathbf{A} \), in a distributed setting, and possibly for OOM scenarios. In this illustration, the distributed HPC system has 5 GPUs among which \( \mathbf{A} \) is distributed (\( \mathbf{A} \) may or may not be cached on GPU), data locality is color coded same as in subsection (V-B), and vertical white lines delineates batch/tile boundaries. The distributed and batched/tilled Gram product \( \mathbf{B} = \mathbf{A}^T \mathbf{A} \) is illustrated in Figure 2a, and the resulting distributed matrix \( \mathbf{B} \) is illustrated in Figure 2b. The distributed matrix \( \mathbf{B} \) is here of size \( n \times n \). It may not be possible to replicate on the different GPUs in OOM scenarios, in which case \( \mathbf{B} \) can be distributed as indicated by the colored rectangles in Figure 2b, showing a possible distribution of \( \mathbf{B} \) among the five different GPUs.

Note that the white and colored lines do not overlap because the number of batches \( n_b \) is not divisible by the number of GPU \( N \). \( \mathbf{B} \) can be computed using \( n_T = n_b \times n_b = 4 \times 4 = 16 \) independent tasks, each with an independent results \( \mathbf{B}_{ij} \) of shape \( b_i \times b_j \) represented by the grey squares forming \( \mathbf{B} \). In OOM-1 scenario, each task \( T_{ij} \) involves a \( H2D(\mathbf{A}_i, \mathbf{A}_j) \) of the local batches \( \mathbf{A}_i \) and \( \mathbf{A}_j \) followed by the computation of \( \mathbf{A}_i^T \), then \( \mathbf{B}_{ij} = \mathbf{A}_i^T \times \mathbf{A}_j \) and subsequently, the copy \( D2H(\mathbf{B}_{ij}) \) back to the host if \( \mathbf{B} \) is not cached on device. Each Task can be queued to a non-default CUDA stream \( Stm_{ij} \) as the tasks are independent, for an asynchronous calculation of \( \mathbf{B} \). Controlling the queue size \( q_s \) allows controlling the number of concurrent tasks running on each GPU, which in turn allows controlling the GPU memory utilization since we can also control the batch size \( b_s \). Further, note that \( \mathbf{B} \) can be computed with \( n_T = 10 < n_T \) fewer tasks if we were to reuse data already uploaded to the GPU: The lower triangular part of \( \mathbf{B} \) is symmetric to the upper triangular part by transposition. Consequently each task \( T_{ij} \) computing non-diagonal \( \mathbf{B}_{ij} \) can save an extra \( H2D(\mathbf{A}_i, \mathbf{A}_j) \), and \( D2H(\mathbf{B}_{ij}) \) by computing the symmetrical \( \mathbf{B}_{ij} = [\mathbf{A}_i^T \times \mathbf{A}_j]^T = \mathbf{A}_j^T \times \mathbf{A}_i \), which is then sent to the GPUs responsible for storing it. Figure 2c illustrates the reduced number of tasks needed to compute distributed \( \mathbf{B} \) ordered in colored and numbered circles, overlaying the respective data segment \( \mathbf{B}_{ij} \) they are responsible for. Each off-diagonal task is also responsible for computing the symmetrical \( \mathbf{B}_{ji} \) not overlaid in the figure. A side effect of using a reduced number of tasks is that tasks now have different execution times; off-diagonal Tasks will run much longer, and in Figure 2d we illustrate (not to scale) the task scheduling for various queue sizes. Each queue has as many CUDA streams as its size \( q_s \), which are ordered along the vertical axis, and the horizontal axis represents execution time. This shows how using CUDA streams helps hide latency by overlapping compute and data copy, as we can see larger queues with more streams execute in a shorter time. Finally, there is an All-reduce communication (AR) between tasks of the same number on the different GPUs to sum the local \( \mathbf{B}_{ij} \) or \( \mathbf{B}_{ij} \) results and to obtain the global results. All (AR) are handled with optimized and low latency NVIDIA communication collectives Library (NCCL) base communicators. The advantage of using NCCL over MPI is discussed in detail by Awan [1].

In the sparse case, where the product of dense factors \( \mathbf{U} \), \( \mathbf{V} \), and \( \mathbf{\Sigma} \) is a significant memory bottleneck even for an out-of-memory implementation, we avoid such computation by performing all the computation at the final stage as detailed in Section IV, which reduces to a series of matrix-vector operation instead of a matrix-matrix operation, which is efficient both computation as well as memory wise. Although pushing the computation at the later stage would require more D2H and H2D communications, these communications are smaller in size in the sparse case while enabling the decomposition of huge sparse matrices. The algorithm is presented in Algorithm 4.

### D. Hardware and computing environment

Benchmark tests were performed on Chicoma, 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 memory. 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. pyDSVD is written in python and uses other off the shelf python libraries such as CuPy [19], Numpy [15], MPI4PY [9] and Scipy [21]. pyDSVD supports both dense and sparse datasets on various hardware architectures.

### VI. BENCHMARKS AND RESULTS

#### A. Scaling benchmarks

![Fig. 3: SVD Scaling results for dense A in (a), and sparse A in (b). Strong scaling results shown with blue solid line, and weak scaling results with brown dashed line.](image)

The first set of benchmark experiments was performed to assess the proposed implementation’s strong and weak scaling characteristics on a distributed heterogeneous system. To this end, we used a dense matrix \( \mathbf{A} \) of shape \((m, n) = [262144, 32768] \) and a sparse matrix of shape \((m, n) = [33554432, 33554432] \) per node. The former had a size \( S_A = 32GB \) in single precision, and the latter a size \( S_A = 4PB \). For the strong scaling experiment in both...
sparse and dense datasets, the largest data that fits in the GPU memory of a node was selected. The sparse matrix was randomly generated with a density $\delta \approx 10^{-6}$ and stored in a sparse Compressed Sparse Row (CSR) format with size $S_0 \approx 12 \text{ GB}$. For the strong scaling, the data shape per $N_n$ node is given as $(m/N_n, n)$, whereas for the weak scaling, each node has the same data shape of $(m, n)$. The scaling experiments were performed for various node counts $N_n = \{1, 2, 4, 8, 16, 32\}$, where each node has 4 GPUs, so the minimum and the maximum number of GPUs utilized were $N = 4$ and $N = 128$, respectively. The largest data size for weak scaling in dense scenario is $32 \times 32 GB = 1 TB$ for 32 nodes whereas for the sparse scenario is $32 \times 4PB = 128 \text{ PB}$ with compressed size of $\sim 32 \times 4 \times 3 = 384 \text{ GB}$. The truncated SVD was evaluated for $k = 32$, and both batch size and queue size were set to 1 making these implementations purely distributed without out of memory feature. Early loop termination in Algorithm 2 (line 10-15) due to convergence is avoided by disabling convergence criterion at lines (13-14) of Algorithm 2. When $A$ is dense, the distributed Gram is directly computed using Algorithm 3, and when $A$ is sparse, Algorithm 4 is used to avoid computing the distributed Gram directly.

Strong and weak scaling benchmark results for dense $A$ are shown in the graph of total execution time vs. number of nodes in Figure 3a. Similarly, the strong and weak scaling benchmark results for sparse $A$ are shown in Figure 3b. Strong scaling results indicate a curve with two distinct slopes, one $N = 1$ to $N = 2$ and the second when $N > 2$. The first slope is due to the lack of inter-node communications when only one node is being used whereas the second slope is the characteristics strong scaling of the distributed systems. Weak scaling results (brown dashed line) for dense and sparse $A$ indicate good scaling maintained up to a node count of $N_n \approx 8$. The performance drop at higher node counts typically indicates latency due to communication becoming increasingly significant at higher node counts. Further, weak and strong scaling results obtained for dense $A$ appear to be better than those obtained with sparse $A$. This is to be expected as Algorithm 3 computes the Gram matrix just once and reuses it in every iteration at line 11 of Algorithm 2. On the other hand, by avoiding computing $A^T A$ with Algorithm 4, communication takes place when performing the two separate All-reduce-sum operations at lines 6 and 8 of Algorithm 4 aggregate with each iteration. Also, note that additional latency resulting from having to batch $V$ in OOM-1 scenarios will affect the performance of the Algorithm 4.

### B. Out of Memory benchmarks

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 4. To this end, the proposed implementation using Algorithm 4 is tested in an OOM-1 scenario, where the same sparse matrix used in the benchmarks above is decomposed up to $k = 32$ with the number of iterations in Algorithm 2 (line 10-15) fixed to 100.

Smaller arrays $A$, $U$, $\Sigma$ are cached on GPU memory, and large co-factor $V$ is stored on the host. This means that during block-operations, $A$ residing on GPU will be tiled, while $V$ residing on the host is batched to GPU. For this experiment, the number of nodes is fixed to $N_n = 2$ to ensure that the algorithm is distributed with inter-node communication taking place and being accounted for in performance evaluation. The execution time of the SVD algorithm and the corresponding peak memory utilization per GPU as a function of number of batches ($n_b = \{2, 4, 8, 16\}$) for various queue sizes ($q_s = \{1, 2, 4, 8\}$) are reported in Figure 4.

In Figure 4a, we show Peak memory utilization vs number of batches for various queue sizes. First, we note that results are reported such that the queue size is at most equal to the number of batches; this is to avoid unnecessary buffer reservation when $q_s > n_b$, which will skew the peak memory utilization. Second, we note a decrease in Peak memory utilization with an increasing number of batches for any fixed queue size, which is expected as increasing the number of batches results in processing smaller batches. Third, we also observe an increased peak memory utilization with increased queue size for any fixed batch number, which we can understand as the aggregated peak memory utilization caused by the different batches. The takeaway is that when an SVD is not possible with given batch size, increasing the number of batches can lower the peak memory per GPU down to levels where SVD becomes feasible. One thing to keep in mind is that ultimately all the curves in Figure 4a will asymptotically decay to a minimum equal to the cumulative sum of the sizes of all arrays cached on GPU. In Figure 4b, we show SVD time vs number of batches for various queue sizes. First, we 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 SVD calculation. This is explained by using large stream queue sizes makes more streams available to 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 $n_b = 8$.  

![Peak Memory Utilization vs Number of Batches](image1)

Fig. 4: (a) OOM SVD peak memory vs number of batches for different queue sizes and (b) OOM SVD time vs number of batches for different queue sizes. The two figures dictate the selection of the right batch size and queue size for faster decomposition, given a peak memory requirement for given data size.
where the SVD time when $q_s = 2$ is lower than when $q_s = 4$ which in turn is lower than when $q_s = 8$. This is explained by the fact that CUDA core counts are finite and that some streams will block and wait when all cores are busy processing other streams, causing load balancing delays. Consequently, it is important to adequately try to fine-tune $q_s$ and $n_s$ for optimal performance.

VII. CONCLUSION

We have presented a distributed out-of-memory implementation of the truncated SVD based on the power method. The potential memory utilization hot spots of the original power method were discussed and found to occur while computing the residual or the Gram matrix. We addressed this concern analytically, redesigning the algorithm by directly updating the singular vectors, eliminating the need to compute both the residual and the gram matrices. Strong and weak scaling results were presented to demonstrate the scalability of the modified algorithm relative to the original implementation. Benchmark results were shown for the case of a sparse matrix of size 128 PB with density $10^{-6}$ compressed to a CSR format of size $\sim 384 \ GB$, decomposed to a rank $k = 32$. The efficacy of batching employed in the proposed implementation to manage peak memory utilization and use of CUDA streams to hide data transfer and communication latency were shown and discussed through the benchmark results.

VIII. ACKNOWLEDGEMENTS

This research was funded by Laboratory Directed Research and Development (201900020DR), and resources were provided by the Los Alamos National Laboratory Institutional Computing Program, supported by the U.S. Department of Energy National Nuclear Security Administration under Contract No. 89233218CNAA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001. The work of Hristo Djidjev has been partially supported by Grant No. BG05M2OP001-1.001-89233218CNA000001.

REFERENCES

[1] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In Proceedings of the 23rd European MPI Users’ Group Meeting, pages 15–22, 2016.

[2] AH Bentbib and A Kamper. Block power method for svd decomposition. Analele Stiintifice ale Universității Ovidius Constanța, Seria Matematica, 23(2):45–58, 2015.

[3] Manish Bhattachari, Gopinath Chennupati, Erik Skau, Raviteja Vangara, Hristo Djidjev, and Boian Alexandrov. Distributed non-negative tensor train decomposition. In 2020 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–10. IEEE, 2020.

[4] Manish Bhattachari, Namita Kharat, Erik Skau, Benjamin Nebgen, Hristo Djidjev, Sanjay Rajopadhye, James P Smith, and Boian Alexandrov. Distributed non-negative rescal with automatic model selection for exascale data. arXiv preprint arXiv:2202.09512, 2022.

[5] Manish Bhattachari, Ben Nebgen, Erik Skau, Makim Eren, Gopinath Chennupati, Raviteja Vangara, Hristo Djidjev, John Patchett, Jim Ahrens, and Boian Alexandrov. pydmf: Python distributed non negative matrix factorization, 2021.

[6] Wajih Halim Boukaram, George Turkyiay, Hatem Ltaief, and David E Keyeyes. Batched qr and svd algorithms on gpus with applications in hierarchical matrix compression. Parallel Computing, 74:19–33, 2018.

[7] Ismael Boureimea, Manish Bhattachari, Makim Eren, Erik Skau, Philip Romero, Stephan Eidenbenz, and Boian Alexandrov. Distributed out-of-memory mmf of dense and sparse data on cpu/gpu architectures with automatic model selection for exascale data. arXiv preprint arXiv:2202.09518, 2022.

[8] Hector Carrillo-Cabada, Erik Skau, Gopinath Chennupati, Boian Alexandrov, and Hristo Djidjev. An out of memory tsvd for big-data factorization. IEEE Access, 8:107749–107759, 2020.

[9] Lisandro Dalcin and Yao-Lung L Fang. mpirpy: Status update after 12 years of development. Computing in Science & Engineering, 23(4):47–54, 2021.

[10] Jack Dongarra. A framework for out of memory svd algorithms. In High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings, volume 10266, page 158. Springer, 2017.

[11] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. The singular value decomposition. Anatomy of optimizing an algorithm for extreme scale. SIAM review, 60(4):808–865, 2018.

[12] Mark Gates, Stanimire Tomov, and Jack Dongarra. Accelerating the svd two stage bidiagonal reduction and divide and conquer using gpus. Parallel Computing, 74:3–18, 2018.

[13] Azzam Haidar, Khairul Kabir, Diana Fayad, Stanimire Tomov, and Jack Dongarra. Out of memory svd solver for big data. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–7. IEEE, 2017.

[14] Per Christian Hansen. Truncated singular value decomposition solutions to discrete ill-posed problems with ill-determined numerical rank. SIAM Journal on Scientific and Statistical Computing, 11(3):503–518, 1990.

[15] Charles R. Harris, K. Jarrod Millman, St´efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Tay- lor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Martin B. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern´andez del R´ıo, Mark Wiebe, Peter Peterson, Pierre G´erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. Nature, 585(7825):357–362, September 2020.

[16] Khairul Kabir, Azzam Haidar, Stanimire Tomov, Aurelien Boutellier, and Jack Dongarra. A framework for out of memory svd algorithms. In International Supercomputing Conference, pages 158–178. Springer, 2017.

[17] Xiang Li, Shusen Wang, Kun Chen, and Zhihua Zhang. Communication-efficient distributed svd via local power iterations. In International Conference on Machine Learning, pages 6504–6514. PMLR, 2021.

[18] Yuechao Lu, Ichitaro Yamazaki, Fumihiko Ino, Yasuyuki Matsushita, Stanimire Tomov, and Jack Dongarra. Reducing the amount of out-of-core data access for gpus-accelerated randomized svd. Concurrency and Computation: Practice and Experience, 32(19):e5754, 2020.

[19] Ryoosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Irido, and Crissman Loomis. 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), 2017.

[20] Eran Rabani and Sivan Toledo. Out-of-core svd and qr decompositions. In PPSC, 2001.

[21] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovskiy, Peurun Peterson, Warren Weckesser, Jonathan Bright, St´efan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Lawson, C J Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17:261–272, 2020.

[22] Stefano Zampini, Wajih Boukaram, George Turkyiay, Omar Knio, and David Keyeyes. HZopus: a distributed memory multi-gpu software package for non-local operators. Advances in Computational Mathematics, 48(3):1–32, 2022.