multiNode:multiGPU implementation of Gram method for out-of-core data access in Randomized SVD algorithms.

Ghanshyam Chandra
Department of Computational and Data Sciences
Indian Institute of Science, Bangalore, India
ghanshyamc@iisc.ac.in

***Abstract*—Singular Value Decomposition (SVD) is a key method for dimensionality reduction in both data science and computational science. SVD is widely used in the computer vision and image processing communities to reconstruct images based on singular values. However, SVD is expanding into other fields, particularly optimization, where it is used for dimensionality reduction while retaining essential features from the original data. As data sizes continue to grow, scaling SVD computations across multiple compute devices and nodes has become increasingly important. Since most scientific computations rely on double-precision arithmetic, and GPUs are continuously improving their double-precision performance, this scaling is crucial. For example, Nvidia V100 GPUs offer 7 teraflops of FP64 compute capability, while A100 GPUs provide nearly 10 teraflops. These capabilities make scaling SVD algorithms on such devices feasible. However, despite their computational power, GPUs are often limited by memory. To manage out-of-core data, algorithms must be designed to handle large data sets and scale across multiple devices and nodes. In this work, we have scaled the Gram method for out-of-core data access across multiple nodes and GPUs. Our solution leverages key GPU features, such as asynchronous memory transfers between host and device, and shared memory utilization for single-GPU computation, which helps avoid warp divergence.**

I. Introduction

Randomized SVD algorithms are popular in the parallel computing community due to their divide-and-conquer approach, which makes them suitable for parallel implementation. However, simply implementing these algorithms is not sufficient, as additional preprocessing steps are required to reduce matrix dimensions while preserving key features of the data. Scaling SVD algorithms necessitates first scaling these preprocessing algorithms.

In this work, we specifically focus on tall and skinny matrices, which are rank-deficient, allowing them to be reduced to square matrices while retaining all the original data's features. Lu et al. [1] proposed various methods for out-of-core data access algorithms, utilizing multi-core architectures. They demonstrated results from several experiments and concluded that the Gram method outperformed other methods in terms of scaling and speedup. However, their implementation was limited to multiple GPUs and multi-core CPUs.

Building upon their work, we extend the Gram method for out-of-core data access across multiple GPUs on multiple nodes. We introduce a novel algorithm for 2D block partitioning of matrices and asynchronous data copying. Additionally, we experimented with shared memory utilization on a single GPU during each iteration to copy local blocks to the global GPU array.

II. Related Work

Lu et al. [1] explored several methods for out-of-core data access. The row-wise 1D partitioning scheme for out-of-core GEMM employs 1D block partitioning of the matrix, followed by block cyclic distribution of data. This scheme assigns blocks to CUDA streams, allowing for data access larger than the device memory size.

The basic FIFO scheme for reducing out-of-core data access uses a divide-and-conquer method. Here, matrices A and P are partitioned across all GPUs, and GEMM is used for local matrix multiplication, with local matrices then stored in a global matrix.

The fused method for reducing out-of-core data access skips the orthogonalization of P and utilizes a fused method. In this approach, two GEMM operations are performed in a single fused loop, and matrix A is transferred only once per iteration. The best-case performance occurs when all blocks are utilized, while the worst-case performance happens when only a single block is used.

The Gram method is based on computing the Gram matrix ($G = A^T A$), which significantly reduces the matrix dimensions for tall and skinny matrices. Power iterations are then applied to this Gram matrix to prevent singular value decay. With more power iterations, more singular values are preserved. Finally, we compute $Q = \text{orth}(GQ)$ and use this preprocessed matrix to compute the SVD.

LAPACK [2] originally implemented the deterministic algorithm proposed by Golub and Kahn [3] sequentially. These algorithms were later implemented in Magma [4], but the implementation is restricted to a single node with multiple GPUs. Nvidia's [5] multi-GPU SVD implementation also supports single-node environments for dense and sparse matrices. Multi-node implementations of SVD algorithms are available through DASK [6] with the RAPIDS library, though these are primarily oriented towards the data science community, focusing on FP32 computation.

III. Methodology

A. Algorithm 1: Gram Method (CPU-GPU communication)

Algorithm 1: Gram Method (CPU-GPU)

```
Result: $C \in \mathcal{R}^{m\times m}$
Input: $A \in \mathcal{R}^{m\times m}$, $B \in \mathcal{R}^{m\times m}$, $C \in 0^{m\times m}$
Global Host Arrays: A, B, C
MPI_Init();
blockSize = m/max_rank;
Assign each MPI_Rank to a GPU.
Local host arrays: h_a, h_b, h_c
MPI_Scatter(A $\rightarrow$ h_A, blockSize)
Local GPU arrays: d_a, d_b, d_c
cudaMemcpyAsync(h_a $\rightarrow$ d_a, h_b $\rightarrow$ d_b)
cudaDeviceSynchronize();
for int i = 0; i < max_rank; i++ do
    if i == 0 then
        gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
        Fill_Mat(ld_c $\rightarrow$ d_c)
    else
        s_ = (i + rank) % max_rank; // Sending rank
        r_ = (max_rank - rank) % max_rank; //
          Receiving rank
        MPI_Sendrecv(h_b, s_ $\rightarrow$ r_);
        cudaMemcpyAsync(h_b $\rightarrow$ d_b);
        cudaDeviceSynchronize();
        gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
        Fill_Mat(ld_c $\rightarrow$ d_c);
    end
end
Do power iteration for count: q
for int z = 0; z < q; z++ do
    for int i = 0; i < max_rank; i++ do
        if i == 0 then
            gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c)
        else
            s_ = (i + rank) % max_rank; // Sending
              rank
            r_ = (max_rank - rank) % max_rank; //
              Receiving rank
            MPI_Sendrecv(h_b, s_ $\rightarrow$ r_);
            cudaMemcpyAsync(h_b $\rightarrow$ d_b);
            cudaDeviceSynchronize();
            gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c);
        end
    end
end
cudaMemcpyAsync(d_c $\rightarrow$ h_c);
cudaDeviceSynchronize();
MPI_Gather(h_c $\rightarrow$ C);
MPI_Finalise();
```

B. Algorithm 2: Gram Method (GPU-GPU communication)

Algorithm 2: Gram Method (GPU-GPU)

```
Result: $C \in \mathcal{R}^{m\times m}$
Input: $A \in \mathcal{R}^{m\times m}$, $B \in \mathcal{R}^{m\times m}$, $C \in 0^{m\times m}$
Global Host Arrays: A, B, C
MPI_Init();
blockSize = m/max_rank;
Assign each MPI_Rank to a GPU.
Local host arrays: h_a, h_b, h_c
MPI_Scatter(A $\rightarrow$ h_A, blockSize)
Local GPU arrays: d_a, d_b, d_c
cudaMemcpyAsync(h_a $\rightarrow$ d_a, h_b $\rightarrow$ d_b)
cudaDeviceSynchronize();
for int i = 0; i < max_rank; i++ do
    if i == 0 then
        gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
        Fill_Mat(ld_c $\rightarrow$ d_c)
    else
        s_ = (i + rank) % max_rank; // Sending rank
        r_ = (max_rank - rank) % max_rank; //
          Receiving rank
        MPI_Send(d_b $\rightarrow$ GDR, s_);
        MPI_Recv(d_b $\rightarrow$ GDR, r_);
        MPI_Barrier();
        gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
        Fill_Mat(ld_c $\rightarrow$ d_c);
    end
end
Do power iteration for count: q
for int z = 0; z < q; z++ do
    for int i = 0; i < max_rank; i++ do
        if i == 0 then
            gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c)
        else
            s_ = (i + rank) % max_rank; // Sending
              rank
            r_ = (max_rank - rank) % max_rank; //
              Receiving rank
            MPI_Send(d_b $\rightarrow$ GDR, s_);
            MPI_Recv(d_b $\rightarrow$ GDR, r_);
            MPI_Barrier();
            gemm(d_c $\rightarrow \langle d\_a \cdot d\_b \rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c);
        end
    end
end
MPI_Gather(d_c $\rightarrow$ GDR, d_C);
if rank == 0 then
    cudaMemcpyAsync(d_C $\rightarrow$ C);
    cudaDeviceSynchronize();
else

end
MPI_Finalise();
```

C. Algorithm 3: Fill_Mat (1D thread blocks)

```
Algorithm 3: Fill_Mat (1D thread blocks)
  Result: $d\_c \in \mathcal{R}^{blockSize \times m}$
  Input: $d\_c \in \mathcal{R}^{blockSize \times m}$,
   $ld\_c \in \mathcal{R}^{blockSize \times blockSize}$
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  for int j = 0; j < m; j++ do
      if $i < m$ then
          d_c[(i + rank) * m + j] = ld_c[i * m + j];
      else
      end
  end
```

D. Algorithm 4: Fill_Mat (2D thread blocks)

```
Algorithm 4: Fill_Mat (2D thread blocks)
  Result: $d\_c \in \mathcal{R}^{blockSize \times m}$
  Input: $d\_c \in \mathcal{R}^{blockSize \times m}$,
   $ld\_c \in \mathcal{R}^{blockSize \times blockSize}$
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if $i < m$ && $j < m$ then
      d_c[(i + rank) * m + j] = ld_c[i * m + j];
  else
  end
```

E. Algorithm 5: Fill_Mat (2D thread blocks with shared memory)

```
Algorithm 5: Fill_Mat (2D thread blocks with shared
 memory)
  Result: $d\_c \in \mathcal{R}^{blockSize \times m}$
  Input: $d\_c \in \mathcal{R}^{blockSize \times m}$,
   $ld\_c \in \mathcal{R}^{blockSize \times blockSize}$
  __shared__ double temp;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if $i < m$ && $j < m$ then
      temp = ld_c[i * m + j];
      d_c[(i + rank) * m + j] = temp;
  else
  end
```

Gram Method Overview:

- Compute Gram Matrix $G = A^T A$
- Perform Power Iteration $Q = GQ$
- Perform QR decomposition $[Q, R] = \text{qr}(Q)$
- Compute SVD of $R : \text{svd}(R) = U\Sigma V^T$
  We are primarily interested in Σ, which contains the singular values on its diagonal.

Algorithm 1 is explained for square matrices but can easily be adapted for rectangular matrices. Assuming 4 GPUs across 2 nodes, the matrix multiplication is partitioned as:

$$C_{ij} = A_i B_j$$

Where A_i and B_j are 2D blocks, with blockSize = $\frac{m}{max\_rank}$.

$$\left[\begin{array}{c|c|c|c} C_{00} & C_{01} & C_{02} & C_{03} \\ \hline C_{10} & C_{11} & C_{12} & C_{13} \\ \hline C_{20} & C_{21} & C_{22} & C_{23} \\ \hline C_{30} & C_{31} & C_{32} & C_{33} \end{array}\right] = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} \left[\begin{array}{c|c|c|c} B_0 & B_1 & B_2 & B_3 \end{array}\right]$$

Here, block A_i multiplied by block B_j gives C_{ij}. The block partitioning over multiple GPUs and nodes is achieved using MPI collective communication through $MPI\_Scatter$. After matrix multiplication, each GPU fills its local matrix d_C, and then MPI_Gather collects the data on the host.

Algorithm 2 follows the same structure as Algorithm 1 but uses GPU Direct RDMA for data transfer and blocking MPI point-to-point communication.

Algorithm 3 is a global CUDA kernel that fills the local matrix C_{00} on each GPU. The extension to Algorithm 4 utilizes 2D thread blocks for improved performance. Algorithm 5 leverages cache locality by utilizing shared memory on the GPU for efficient data transfer, avoiding warp divergence and ensuring coalesced memory access.

IV. Experiments and Results

A. Experiment Setup

Resources Available:

- 2 NUMA nodes with 4 V100 GPUs.
- Computations are performed in double precision.

Experiments are carried out using 2 V100 GPUs on a single node.

B. Experiment 1 (Algorithm 1 + Algorithm 5)

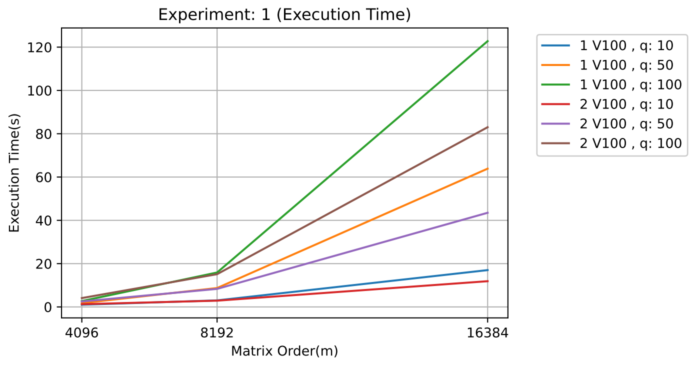

Fig. 1. Execution Time (s) vs. Matrix Order.

C. Experiment 2 (Algorithm 1 + Algorithm 4)

D. Experiment 3 (Algorithm 1 + Algorithm 3)

E. Experiment 4 (Algorithm 2 + Algorithm 5)

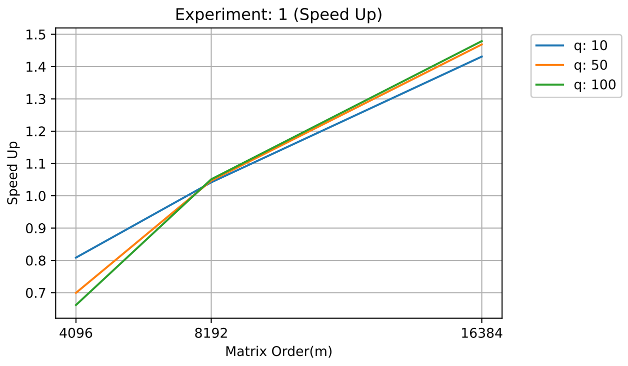

Fig. 2. Speedup.

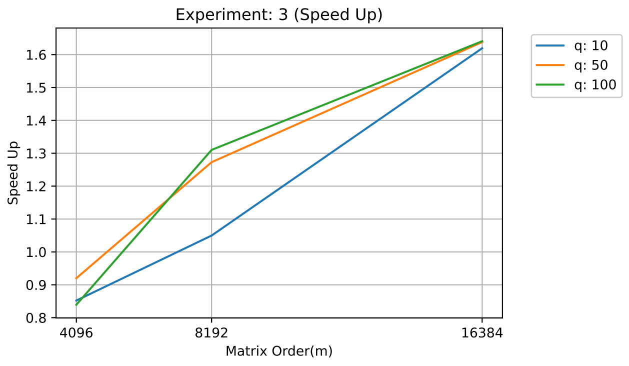

Fig. 6. Speedup.

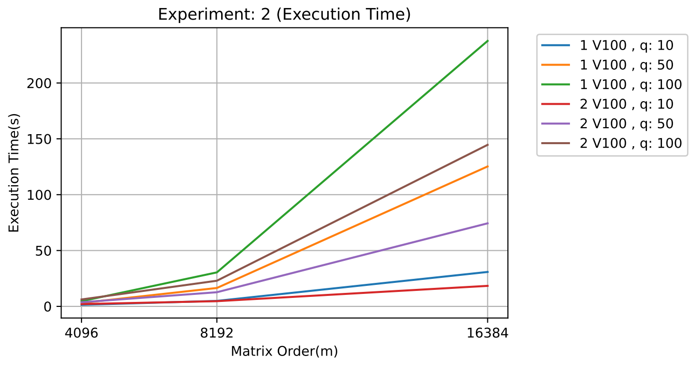

Fig. 3. Execution Time (s) vs. Matrix Order.

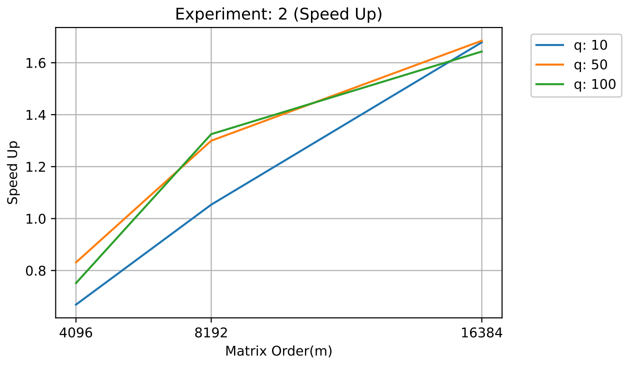

Fig. 4. Speedup.

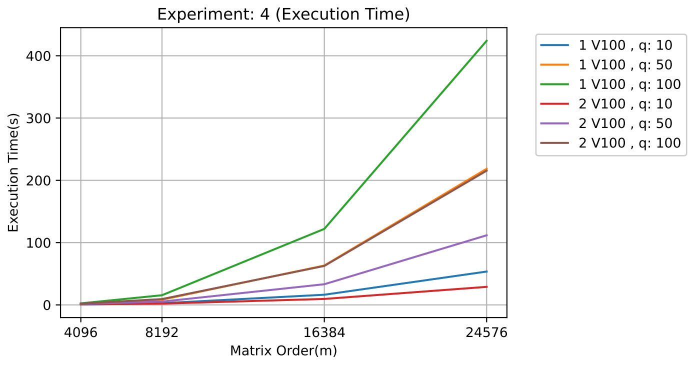

Fig. 7. Execution Time (s) vs. Matrix Order.

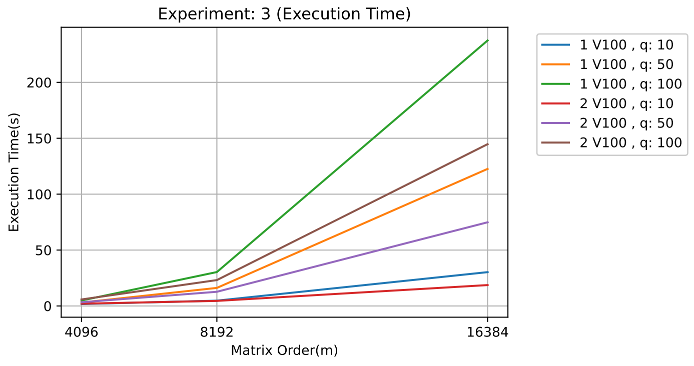

Fig. 5. Execution Time (s) vs. Matrix Order.

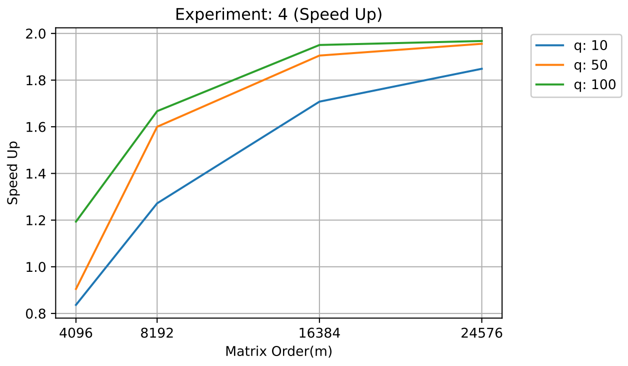

Fig. 8. Speedup.

F. Experiment 5 (Algorithm 1 and Algorithm 2 + Algorithm 5)

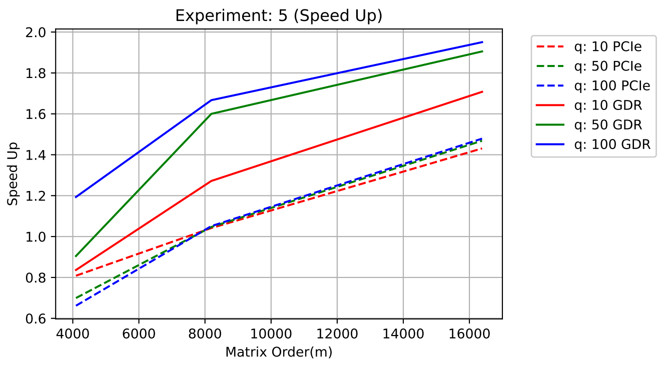

Fig. 9. Speedup.

G. Results

- Experiment 4 clearly shows the highest speedup due to direct GPU-to-GPU memory transfers.
- Experiment 5 compares the scalability of Algorithm 1 and Algorithm 2. It is evident that Algorithm 2 is more scalable, achieving nearly 2x speedup with 2 V100 GPUs, which approaches the ideal case, particularly with larger problem sizes.
- From Figures 1, 3, and 5, it is clear that Algorithm 5, which leverages shared memory to exploit cache locality, outperforms the other two algorithms. Even though the other two algorithms differ in implementation (one using 1D thread blocks and the other 2D thread blocks), their execution time and speedup are nearly identical.
- **Weak Scaling:**
  From Experiment 1, it is evident that when the problem size is small (up to order 8192), both single-GPU and dual-GPU setups have similar execution times. However, at matrix size 16384, keeping the problem size constant and increasing the number of compute devices results in improved speedup and reduced execution time, indicating weak scaling. The scaling may improve further as the problem size increases.
- **Strong Scaling:**
  Across all speedup plots, we observe that increasing the matrix order or problem size results in higher speedups, demonstrating the strong scaling of the algorithm.

V. Conclusions

Algorithm 2 is more scalable than Algorithm 1 due to the lower overhead of memory transfers and increased communication speed over Mellanox Infiniband connections.

This work proposes a novel algorithm and evaluates its scalability, although the results have not yet been verified against baseline methods. Therefore, at this stage, it remains a black box. After successful verification with the baseline methods, we can conclude that this algorithm contributes the following:

- A novel algorithm for the multi-node extension of the Gram method.
- Asynchronous data transfer throughout all host-to-device communication.
- An algorithm based on direct GPU-to-GPU communication.

Future Work

- Verification with standard results.
- Introduction of mixed-precision computation.
- Extending this work to heterogeneous architectures that leverage both many-core processors and multi-GPU setups across multiple nodes.

References

[1] Y. Lu, I. Yamazaki, F. Ino, Y. Matsushita, S. Tomov, and J. Dongarra, "Reducing the amount of out-of-core data access for GPU-accelerated randomized SVD," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 19, Apr. 2020. [Online]. Available: https://doi.org/10.1002/cpe.5754

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[3] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, vol. 2, no. 2, pp. 205–224, Jan. 1965. [Online]. Available: https://doi.org/10.1137/0702016

[4] B. J. Smith, "R package magma: Matrix algebra on gpu and multicore architectures, version 0.2.2," August 27, 2010, [On-line] http://cran.r-project.org/package=magma.

[5] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 11.3," 2021. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[6] Dask Development Team, *Dask: Library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org