mutiNode: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*—SVD is key for any Data science or Computational science dimensionality reduction. SVD is particularly used in computer vision and image processing community to reconstruct image based on singular values, however day by day SVD is expanding to other fields too, particularly in optimisation fields where people are using SVD for dimensionality reduction and use the same features as of fully available data, considering increase in data size required to be processed with SVD algorithms we are in need to scale up SVD computation over multiple compute devices and multiple nodes. Since most of the scientific computation involves double precision computation and day by day GPU's are increasing their compute capability and hence increasing double precision performance Nvidia V100 GPU's has 7 Tera Flop's of FP64 compute capability and A100 has almost 10 Tera Flop's of FP64 compute capability these makes us to scale SVD algorithms to these devices but although these devices can do very fast computation but are limited by it's memory, hence in order to handle out of core data we need to design algorithms which can handle out of core data as well as can scale over multiple devices across different nodes. In current work we have scaled Gram method of out of core data access over multiple nodes and multiple GPU's exploiting the core features of GPU such as asynchronous memory copy between Host to Device and shared memory utilisation for single GPU computation to avoid warp-divergence condition.**

I. Introduction

Randomised SVD algorithms are popular in parallel community due to it's divide and conquer approach, which makes it suitable for parallel implementation, but even though we have these algorithms we can't just implement these algorithms and do computation because we need to further reduction of these matrices in order to reduce their dimension, such algorithms reduces matrix dimension preserving all the feature of data hence to scale up SVD algorithms we need to scale up these pre-processing algorithms first. Here in this work we have particularly focused on tall and skinny matrices which are rank deficient and hence can be converted into square matrices preserving all the feature of initial data. Lu et al. [1] has proposed different methods for these out of core data access algorithms in which they have utilised multi-core architecture to implement these algorithms and showed results with different experiments, their work concludes that Gram method has particularly outperformed all other methods in terms of scaling and speed-up but implementation is only restricted to multiple GPU's and multi-core CPU's, considering their implementation as base we extended Gram method for out of core data access in multiple GPU's on multiple Nodes with novel algorithm for 2D block partitioning of matrices and asynchronous data copy, we also experimented with shared memory utilisation on single GPU on each iteration to copy local blocks to global GPU array.

II. Related Work

Lu et al. [1] has experimented with different methods to access out of core data. Row-wise 1D partitioning scheme for out of core GEMM utilises 1D block partitioning of matrix followed by block cyclic distribution of data in which blocks are assigned to cuda streams hence block cyclic distribution allows accessing data which are larger then the device memory size.

Basic FIFO scheme for reducing out of core data access utilises divide and conquer method in which matrix A and P are partitioned across all the GPU's and utilising GEMM for local matrix multiplication and then then storing local matrices into global matrix.

Fused Method for reducing out of core data access skips orthogonalisation of P and utilises fused method in which two GEMM operations are performed on single fused loop, matrix A is transferred only once in each iteration here the best case is obtained when all the blocks are utilised and worst case will be when only single block is utilised.

Gram method is based on computation of gram matrix ($G = A^T A$) which tremendously reduces matrix dimension for tall and skinny matrices and then we compute power iteration on this Gram matrix to prevent singular decay, with more power iteration we preserve more singular values. So at last we compute $Q = orth(GQ)$ then further use this pre processed matrix to compute SVD.

LAPACK [2] initially implemented deterministic algorithm proposed by Golub and Kahn [3] sequentially and then these algorithms are implemented in Magma [4] but are restricted on single node on multiple GPU's, Nvidia [5] has single node multi GPU implementation of SVD for dense and sparse matrices. multi-node implementation of SVD algorithms is available with DASK [6] with RAPIDS library but these are specifically oriented towards data science community who deals data mostly for FP32 computation.

III. Methodology

A. Algorithm - 1 Gram Method (2D block Partioning)

Algorithm 1: Gram Method (CPU-GPU)

```
Results: C ∈ $\mathcal{R}^{m\times m}$.
Input: A ∈ $\mathcal{R}^{m\times m}, B \in \mathcal{R}^{m\times m}, C \in 0^{m\times m}$
Global Host array A, B, C
MPI_Init();
blockSize = m/max_rank
Assign each MPI_Rank to each GPU
 local host array: h_a, h_b,h_c
MPI_Scatter(A$\rightarrow h_A, blockSize$)
local GPU array: d_a , d_b ,d_c
cudaMemcpyAsync( h_a $\xrightarrow{PCIe}$ d_a, h_b $\xrightarrow{PCIe}$ d_b)
cudaDeviceSynchronize();
for (int i = 0; i < max_rank;i++) do
    if (i==0) then
        gemm(d_c$\rightarrow \langle$d_a.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_ $\xrightarrow{PCIe}$ r_);
        cudaMemcpyAsync( h_b $\xrightarrow{PCIe}$ d_b)
        cudaDeviceSynchronize();
        gemm(d_c$\rightarrow \langle$d_a.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.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_ $\xrightarrow{PCIe}$ r_);
            cudaMemcpyAsync(h_b $\xrightarrow{PCIe}$ d_b)
            cudaDeviceSynchronize();
            gemm(d_c$\rightarrow \langle$d_a.d_b$\rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c)
        end
    end
end
cudaMemcpyAsync( d_c $\xrightarrow{PCIe}$ h_c);
cudaDeviceSynchronize();
MPI_Gather(h_c $\rightarrow$ C);
MPI_Finalise();
```

B. Algorithm - 2 Gram Method (2D block Partioning)

Algorithm 2: Gram Method (GPU-GPU)

```
Results: C ∈ $\mathcal{R}^{m\times m}$.
Input: A ∈ $\mathcal{R}^{m\times m}, B \in \mathcal{R}^{m\times m}, C \in 0^{m\times m}$
Global Host array A, B, C
MPI_Init();
blockSize = m/max_rank
Assign each MPI_Rank to each GPU
 local host array: h_a, h_b,h_c
MPI_Scatter(A$\rightarrow h_A, blockSize$)
local GPU array: d_a , d_b ,d_c
cudaMemcpyAsync( h_a $\xrightarrow{PCIe}$ d_a, h_b $\xrightarrow{PCIe}$ d_b)
cudaDeviceSynchronize();
for (int i = 0; i < max_rank;i++) do
    if (i==0) then
        gemm(d_c$\rightarrow \langle$d_a.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 $\xrightarrow{GDR}$ s_);
        MPI_Recv(d_b $\xrightarrow{GDR}$ r_);
        MPI_Barrier();
        gemm(d_c$\rightarrow \langle$d_a.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.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 $\xrightarrow{GDR}$ s_);
            MPI_Recv(d_b $\xrightarrow{GDR}$ r_);
            MPI_Barrier();
            gemm(d_c$\rightarrow \langle$d_a.d_b$\rangle$)
            Fill_Mat(ld_c $\rightarrow$ d_c)
        end
    end
end
MPI_Gather(d_c $\xrightarrow{GDR}$ d_C);
if (rank==0) then
    cudaMemcpyAsync( d_C $\xrightarrow{PCIe}$ C);
    cudaDeviceSynchronize();
else

end
MPI_Finalise();
```

C. Algorithm - 3 Fill_Mat

Algorithm 3: Fill_Mat (1D thread blocks)

```
Results: 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

Algorithm 4: Fill_Mat (2D thread blocks)

```
Results: 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

Algorithm 5: Fill_Mat (2D thread blocks with utilising cache locality of shared memory)

```
Results: 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:

- Compute Gram Matrix(G)
$$G = A^T A$$
- Do Power Iteration.
$$Q = GQ$$
- Do QR
$$[Q, R] = qr(Q)$$
- Compute SVD
$$svd(R) = U \Sigma V^T$$

Here we are interested in Σ which contains singular value on it's diagonal element.

Algorithm 1 is explained for square matrices which can easily be changed to accommodate rectangular matrices assuming there are 4 GPU's in 2 Nodes.

$$C_{ij} = A_i B_j \tag{1}$$

Here A_i and B_j are 2D blocks of matrices partitioned from from original matrix with $blockSize = \frac{m}{max\_rank}$

$$\left[\begin{array}{c|c|c|c} C_{00} & C_{01} & C_{02} & C_{03} \\ \hline C_{10} & C_{11} & C_{12} & C_{13} \\ \hline C_{20} & C_{21} & C_{22} & C_{23} \\ \hline C_{30} & C_{31} & C_{32} & C_{33} \end{array}\right] = \left[\begin{array}{c} A_0 \\ A_1 \\ A_2 \\ A_3 \end{array}\right] \left[\begin{array}{c|c|c|c} B_0 & B_1 & B_2 & B_3 \end{array}\right] \tag{2}$$

Where

$$\left[\begin{array}{c} C_0 \\ C_1 \\ C_2 \\ C_3 \end{array}\right] = \left[\begin{array}{c|c|c|c} C_{00} & C_{01} & C_{02} & C_{03} \\ \hline C_{10} & C_{11} & C_{12} & C_{13} \\ \hline C_{20} & C_{21} & C_{22} & C_{23} \\ \hline C_{30} & C_{31} & C_{32} & C_{33} \end{array}\right]$$

Here block A_i multiplied with block B_j will give us C_{ij}. Now to implement 2D block partitioning over multiple GPU's across the nodes we will use MPI collective communication $MPI\_Scatter$ to divide matrix on blocks now we will copy these local host arrays to each rank which is assign to single GPU and do matrix multiplication, after each multiplication we fill the local C matrix d_C on each GPU an then after block cyclic execution we collect data on host with $MPI\_Gather()$.
Algorithm 2: is same as of Algorithm 1 except data transfer is taking place through GPU Direct RDMA.
with blocking MPI point to point communication.
Algorithm 3: is global CUDA kernel implemented to fill the block-wise local C matrix e.g. C_{00} on local C matrix e.g. C_0 on each GPU in each block cyclic pass, this algorithm uses 1D thread block hence each for loop is running on each thread since we have threads i y dimension also we can extend our Algorithm 3 to Algorithm 4 where we are utilising the thread blocks in both the dimensions, now again we are missing something on optimisation and that is exploiting cache locality with the use of shared memory present in GPU, for V100 it's only 400KB which can only handle handle few thousand FP64 elements of matrices and since we are only transferring the data and not doing any further computation on the Fill_Mat kernel hence going for coalesced memory access to avoid warp divergence is not required. here we are increasing cache locality for each thread to first store data in shared memory and then transfer it to other memory address within GPU and that is what implemented in Algorithm 5.

IV. Experiments and Results

A. Experiment Setup

Resources available

- 2 NUMA nodes with 4 V100 GPU.
- computation are performed with double precision.

Experiments are carried out with 2 V100 GPU on single Node.

B. Experiment 1 (Algorithm 1 + Algorithm 5)

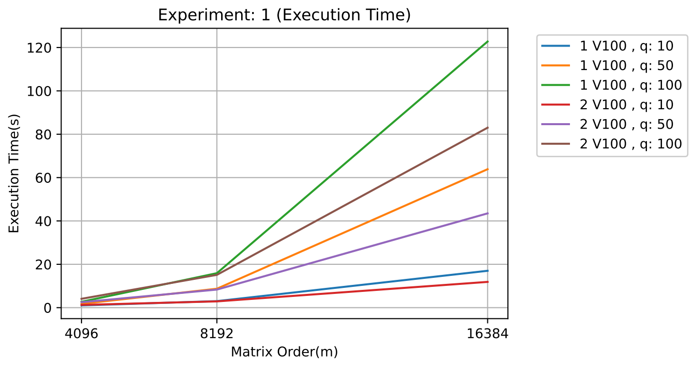

Fig. 1. Execution Time(s) v/s Matrix order.

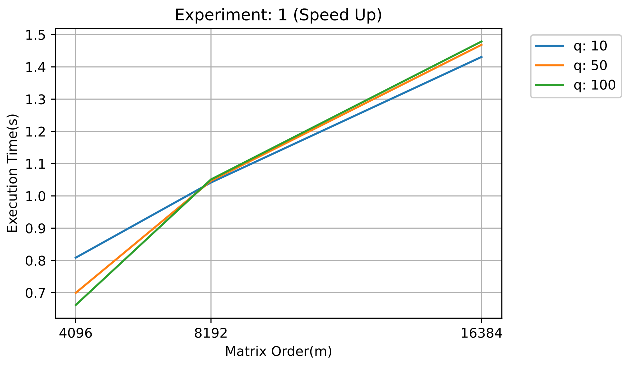

Fig. 2. Speed Up

C. Experiment 2 (Algorithm 1 + Algorithm 4)

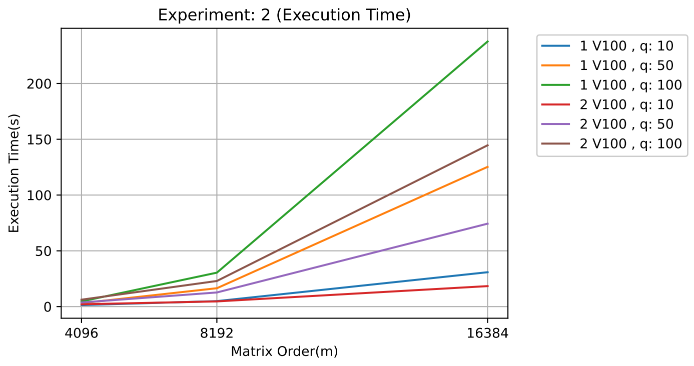

Fig. 3. Execution Time(s) v/s Matrix order.

D. Experiment 3 (Algorithm 1 + Algorithm 3)

E. Results

- From Fig: 1 , Fig: 3 and Fig: 5 it's clear that Algorithm: 5 which uses shared memory to exploit cache locality outperforms two other algorithms, while the other two algorithms are performing almost similarly even though both are completely different one uses 1D thread block while other uses 2D thread blocks but execution time as well as speed-up's are almost same.

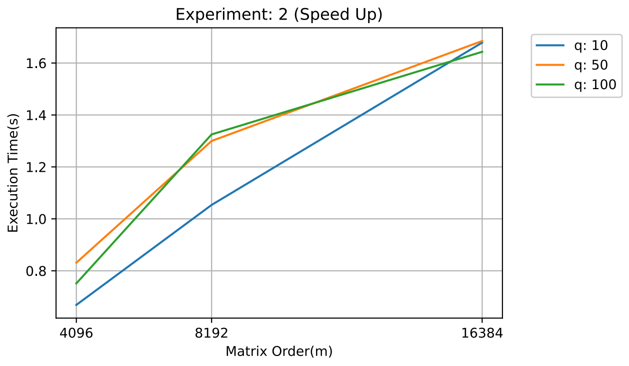

Fig. 4. Speed Up

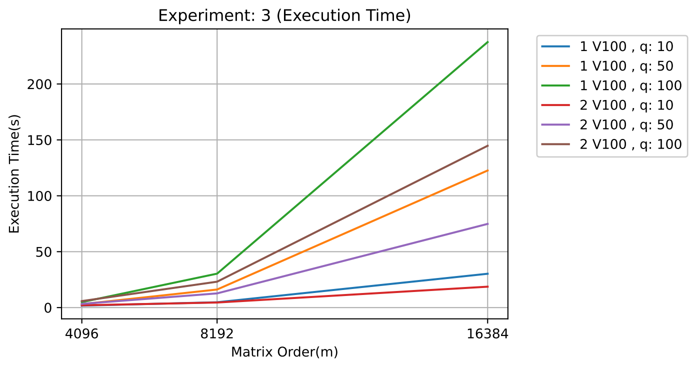

Fig. 5. Execution Time(s) v/s Matrix order.

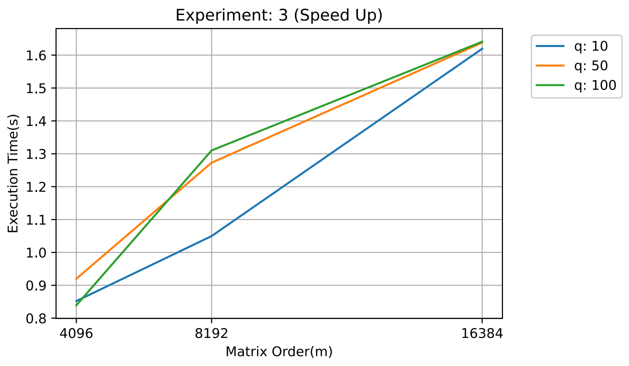

Fig. 6. Speed Up

- **Weak Scaling**
  From experiment 1 it's clearly visible that keeping the problem size small upto order 8192 both single GPU and dual GPU both have similar execution time, but at 16384 keeping the problem size constant and increasing no. of compute devices speed up increases on execution time decreases which directly signifies that algorithm is weak scalable and may be with increase in order scaling

increases more.

- **Strong Scaling**
  From all speed-up plots we can clearly see that with increase in matrix order or problem size the speed up increases which signifies Strong scaling of algorithm.

V. Conclusions

This work proposes novel algorithm an it's implementation scalability test but still we have not verified verified results with base methods hence till now it's a black box, after successful verification with base paper we can conclude that this algorithm has contributed:

- Novel algorithm for multi-node extension of Grams method.
- Asynchronous data transfer throughout all Host to Device data transfer.
- Algorithm based on GPU-GPU direct communication.

Future Work

- Verification with standard results.
- Introducing mixed precision computation.
- Extending this work to heterogeneous architecture which which leverage many-core processor as well as multiGPU across multiNode.

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