# Parallelizing SVDs over Multiple GPUs

**David Kott**
Department of Mathematics
david.kott@colostate.edu

**Connor Price**
Department of Mathematics
connor.price@colostate.edu

## 1 Introduction and Motivation

Singular Value Decomposition (SVD) is resource-intensive, particularly when dealing with large matrices. By leveraging multiple GPUs, we can handle and analyze larger datasets in real time. So, the primary motivation for parallelizing SVDs is scalability. SVD is crucial for training large and intricate models in machine learning applications. Through parallelization, we can expedite and increase the frequency of training. This project is directly applicable to numerous fields, including our work in the Pattern Analysis Lab under the guidance of Dr. Kirby. We are utilizing GPUs as our target platform and using cuBLASXt API for multiple GPUs. This approach allows us to focus on the high-level structure of our algorithms while the compiler handles the GPU programming. We also plan to use nvprof and Nsight to profile our parallelization, provided below.

## 2 DGX2

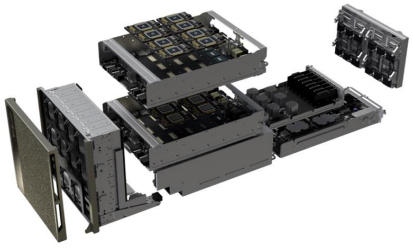

Figure 3: DGX2

All computational testing is conducted on the Mathematics Department's cluster "Slipstick". This cluster includes a node called as DGX2. The DGX2 has 16 V100 GPUs and a unique tool, NVLink. NVLink facilitates exceptionally fast communication between the GPUs, effectively addressing a significant bottleneck in such systems.

In addition to these features, the DGX2 is powered by 1.5 terabytes of RAM and 96 threads on the host. This robust configuration allows for a theoretical maximum computational capacity of 2 teraflops, providing substantial power for complex calculations and data processing tasks. This makes the DGX2 an incredibly efficient and powerful tool for computational testing.

## 3 Techniques

This implementation of SVD deploys QR decomposition in C++. The function $svd$ takes a matrix $A$ of size $m$ by $n$ as input and outputs the singular vectors $U$ and $V$ and singular values $S$. We are allocating our matrices dynamically in 1D arrays. We knew one of the biggest challenges was memory resources for the matrices, so we elected to use arrays instead of vectors.

We have constructed the following Matlab pseudocode, which we translated to C++. This pseudocode is in the Appendix.

### 3.1 OpenACC

In the initial stages of our project, we chose OpenACC as our method for parallelizing computations. OpenACC is a directive-based parallel programming model. It's designed for systems with separate memory spaces.

For matrix multiplication operations, we used OpenACC's $gang$ and $worker$ clauses. These clauses distribute loop iterations among CUDA blocks and threads, respectively. This is essentially a form of CUDA coalescing implemented within the OpenACC framework. The $gang$ clause is used to distribute the iterations of the loop across $gangs$ (which can be considered equivalent to CUDA blocks). The $worker$ clause is used to distribute the iterations of the loop across $workers$ within a $gang$ (which can be considered equivalent to CUDA threads within a block).

From this point forward, the parallelism in our code was consistent with what you would see in an equivalent OpenMP implementation. OpenMP,

like OpenACC, is a directive-based programming model designed for shared memory systems. Therefore, most of our pragma calls were fairly intuitive, given the similarities between the two.

However, we encountered performance issues with OpenACC. The overhead associated with initializing kernels (the functions that are executed on the GPU) proved to be too costly. Kernel initialization involves setting up the execution configuration and transferring data from the CPU to the GPU. This overhead was significant enough to severely limit the size of the matrices we could work with.

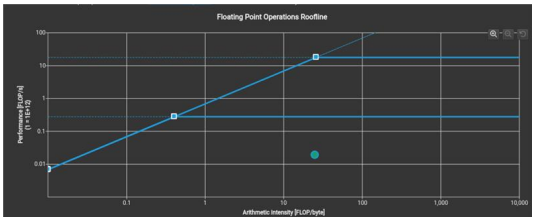


Figure 4: Roofline for OpenACC across a single GPU. Achieved .023 TerraFLOPS

### 3.2 CuSolverMG

Following the performance results with OpenACC, we focused on utilizing CuBLAS's CuSolverMG and CuBLASXt for our computations. This section will talk about cuSolverMG. However, both of these libraries are provided by NVIDIA for GPU-accelerated applications.

CuBLAS is a GPU-accelerated version of the complete standard BLAS library, which stands for Basic Linear Algebra Subprograms. BLAS is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. In our case, CuBLAS handled simple matrix operations efficiently on the GPU.

To parallelize the symmetric eigendecomposition, specifically the $A^T A$ operation, we employed CuSolver. CuSolver is another library provided by NVIDIA, but unlike CuBLAS, it is designed to handle more complex operations like decompositions. Eigen-decomposition is a type of decomposition that involves breaking a matrix down into its constituent parts to make certain operations simpler and more numerically stable. CuSolverMG, on the other hand, is the multi-GPU version of CuSolver. It is designed to leverage multiple GPUs to perform linear algebra operations more quickly and efficiently. This is particularly useful for large-scale problems that may not fit into the memory of a single GPU or for computations that can be sped up by using multiple GPUs.

In our implementation, we utilized $cuSolverMgHandlet$, a type provided by the CuSolver-MG library. This handle maintains resources for the multi-GPU routines. It's essentially a context used by the CuSolver-MG library to manage the resources needed for its operations. To create this context, we used the function $cuSolverMgCreate$. This function initializes a cuSolver multi-GPU context, which is then used for subsequent calls in the library. In our code, $cuSolverMgH$ is an instance of this context. We employed CuSolver in the second step of our Singular Value Decomposition (SVD) process. Specifically, we used it to calculate the eigenpairs of the matrix $A^T A$ using $cuSolverMgSyevd$. This function computes the eigenvalues and eigenvectors of a symmetric matrix, which is exactly what we need for this step of the SVD.

The $cusolverMgH$ handle plays a crucial role in this process. It manages the resources needed for the multi-GPU routines in the second step. We first set the list of devices to use with the function $cusolverMgDeviceSelect$. This function lets us specify which GPUs we want to use for our computations.

Once the devices are selected, we use the handle in calls to $cusolverMgSyevd\_bufferSize$ and $cusolverMgSyevd$. The former function is used to compute the size of the work buffer needed for the latter function, which then computes the eigenvalues and eigenvectors.

After the computation is complete, we destroy the handle with the function $cusolverMgDestroy$. This is an important step as it frees up the allocated resources for the handle, preventing memory leaks and ensuring our program runs efficiently.

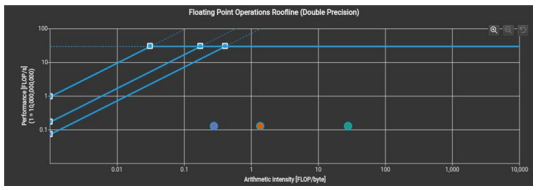


Figure 5: Roofline for CuSolver-MG across two GPUs. Achieved .11 TerraFLOPS

### 3.3 CuBLASXt

In addition to cuSolver-MG, we used cuBLASXt. The cuBLASXt library is designed for performing Basic Linear Algebra Subprograms (BLAS) operations on multi-GPU systems, while the cu-

SOLVER library provides a suite of dense and sparse direct solvers.

CuBLASXt is used to calculate the multiplication of matrix $A^T A$. The operation is executed using the function $cublasXtSgemm$. The 'Sgemm' in the function name stands for 'single-precision general matrix-matrix multiplication.' This function performs the multiplication of two matrices in single-precision arithmetic. The function takes several parameters, including the handles to the cuBLASXt context, the operation types (whether to perform a transpose or a conjugate transpose), the dimensions of the matrices, and the matrices themselves, among others. It then performs the matrix multiplication operation on the GPU, using its parallel processing capabilities to perform the operation efficiently.

The type $cublasXtHandle_t$ represents a handle to the cuBLASXt context. This handle is crucial as it maintains the resources needed for the multi-GPU routines. An instance of this handle, denoted as $handle_xt$, is created and initialized with the function $cublasXtCreate$. This function is responsible for setting up the cuBLASXt context.

Following the creation and initialization of the handle, the function $cublasXtDeviceSelect$ is employed. This function sets the list of devices that the multi-GPU context will use. In the given context, it is set to utilize all available GPUs.

After the computation is completed, the resources allocated for the handle are freed up. This is done using the function $cublasXtDestroy$. This function destroys the handle, ensuring efficient memory management and preventing potential memory leaks. This entire process allows for efficient parallel computations on multiple GPUs, significantly improving the performance of matrix operations.

# 4 Performance

To analyze the performance of our code, we used Nsight Compute. This program lets users obtain feedback on their CUDA kernels, including generated speed-up recommendations. All metrics are calculated through Speed of Light (SOL), Nvidia's metrics for how close a kernel is to achieving theoretical maximum performance. Within the SOL metrics is the option for a roofline visualization which was the focus of our analysis.

The OpenACC and CuBLASXt/CuSolverMG code created over a thousand kernels, each with its own SOL metrics. Nearly all these kernels ran for less than a millisecond, so the focus was on the symmetric eigen decomposition, the most significant portion of our code regarding duration.

The OpenACC version of singular value decomposition achieved .023 TerraFLOPS when running a 100 by 100 4. It is worth noting this was the largest size we could run within a reasonable time frame, and it was performed on one device, so communication wasn't nearly as significant as later work.

Using Nvidia's packages for matrix multiplication and eigen decomposition increased flops to around .11 TerraFLOPS 5.

Runtime benchmarks were done for 100x100 matrices. OpenACC ran in about 5 seconds, whereas the CublasXt/CuSolverMG ran in about .4 seconds. It is worth noting that a significant portion of the .4 second was initialized, so scaling the matrix would have relatively little impact on runtime until sizes of about 1,000 1,000 are tested.

Further tests were done on the CublasXt/CuSolverMG SVD code to examine the effects of increasing GPUs. To our surprise, we noticed a consistent slowdown of about 3 seconds when a GPU was added (tested with a 50,000 by 20,000 matrix). This indicated that increasing GPUs should only be done if more DRAM is needed to accommodate the problem.

# 5 Applications

SVD is a powerful mathematical tool with broad applications ranging from solving complex mathematical problems to powering modern machine learning algorithms and systems. Its ability to decompose matrices into simpler constituents makes it an invaluable technique in many fields. It is used to solve linear equations, compute matrix inverses, and calculate matrix ranks. It's also used for Principal Component Analysis (PCA), a technique for reducing the dimensionality of datasets, while preserving as much variance as possible. It is also used in in various data analysis and signal processing tasks.

## 5.1 Diffusion Models for Image Generation

In our work in the Pattern Analysis Lab, we are currently looking at generating new photorealistic images from the CelebA dataset. Singular Value Decomposition plays a crucial role in Diffusion models for image generation, including the generation of faces.

Diffusion models are a class of generative models that generate new samples by simulating a stochastic process, where a data point is gradually transformed into a random variable through a series of small random diffusion steps. The generation process is reversed to transform this random variable back into a data point.

In the context of image generation, SVD can be used in the preprocessing stage, where high-dimensional image data is often decomposed into a lower-dimensional representation. This is particularly useful when working with large datasets of images, such as faces, where each image can be considered as a high-dimensional vector. SVD helps to extract the most important features from these images while reducing the dimensionality of the data.

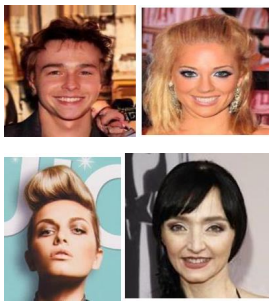

Figure 6: Above: Training images from CelebA dataset. Below: Generated images from diffusion model

## 6 Conclusion

A significant challenge with large scale GPU linear algebra computations is DRAM on devices. A single V100 GPU has 32 Gigabytes of RAM making float matrices larger than 40,000 40,000 impossible. In the past we've resorted to using high memory CPU computing nodes which resulted in runtime that were unreasonable for the amount of singular value decompositions we wanted to run. We know not only have a program which can compute up to a 120,000 120,000 singular value decomposition on the DGX2 but have applied the code to a current research problem within our field.

## 7 Future Work

For potential tasks in the future, we aim to significantly enhance the usability and accessibility of our software by transforming the existing C++ codebase into a Python package. This strategic move will not only streamline the deployment process but also broaden the user base. Additionally, we plan to conduct an in-depth examination of alternative data types, with a particular focus on half precision, to optimize computational performance and efficiency. A critical component of our future work involves a comparative performance analysis across various GPU architectures. This will enable us to pinpoint the most effective configurations for different computational tasks. Lastly, we anticipate expanding the scope of our project to include matrix inversions, thereby increasing the versatility and applicability of our software in solving a wider array of problems.

# 8 Appendix

## 8.1 Tables and Graphs

Here is our speedup for the $50000x20000$ matrix on $1 - 16$ GPUs.

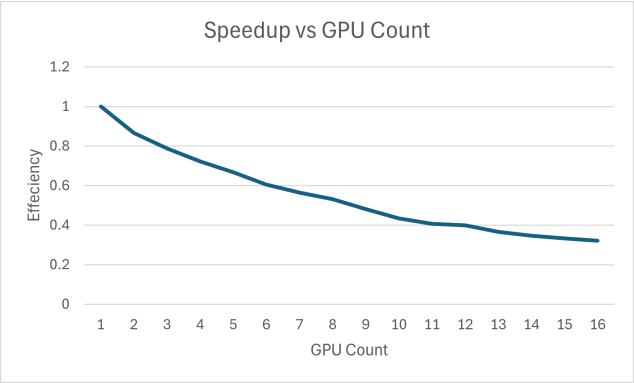


Figure 7

| Runtime | Speedup | Effeciency |
|---|---|---|
| 26 | 1 | 1 |
| 30 | 0.866667 | 0.433333 |
| 33 | 0.787879 | 0.262626 |
| 36 | 0.722222 | 0.180556 |
| 39 | 0.666667 | 0.133333 |
| 43 | 0.604651 | 0.100775 |
| 46 | 0.565217 | 0.080745 |
| 49 | 0.530612 | 0.066327 |
| 54 | 0.481481 | 0.053498 |
| 60 | 0.433333 | 0.043333 |
| 64 | 0.40625 | 0.036932 |
| 65 | 0.4 | 0.033333 |
| 71 | 0.366197 | 0.028169 |
| 75 | 0.346667 | 0.024762 |
| 78 | 0.333333 | 0.022222 |
| 81 | 0.320988 | 0.020833 |

Figure 8: Speedup & Efficiency over 16 GPUs

## 8.2 Matlab Rubric

Here is the Matlab pseudocode that we are emulating:

```
A = rand(5,5);
[U, S, V] = raw_svd(A);
norm(A-U*diag(S)*V')
function [U, S, V]=raw_svd(A)
    [V,S] = eigen_decomposition(A'*A);
    S = sqrt(diag(S));
    U = A*V*diag(1./S);
end

function [V, D] = eigen_decomposition(A)
    V = eye(size(A));
    D = A;
    tol = 1e-4;
    max_iter = 1000;
    for iter = 1:max_iter
        [Q, R] = qr_decomp(D);
        D = R * Q;
        V = V * Q;
```

```
        off_diagonal = D - diag(diag(D));
        if norm(off_diagonal)* norm(off_diagonal)< tol
            break;
        end
    end
    % Extract eigenvalues
    D = diag(diag(D));
end

function [Q, R] = qr_decomp(A)
    [m, n] = size(A);
    Q = zeros(m,n);
    R = zeros(n,n);
    for k=1:n
        R(k,k) = norm(A(1:m,k),2);
        Q(1:m,k) = A(1:m,k)/R(k,k);
        for j = k+1:n
            R(k,j) = Q(1:m,k)'*A(1:m,j);
            A(1:m,j)=A(1:m,j)-Q(1:m,k)*R(k,j);
        end
    end
end
```