Parallelization of Singular Value Decomposition

Sibi Raja[1], Eliel Dushime[1], Matthew Villaroman[1], and Bowen Zhu[1]

[1]Harvard University, John A. Paulson School of Engineering and Applied Sciences

May 6, 2023

Abstract

The Singular Value Decomposition (SVD) is a fundamental matrix operation used in a wide range of scientific and engineering applications. The computation of SVD involves several linear algebra operations, such as the dot product, matrix multiplication, and matrix transposition, due to the nature of working with vectors and matrices. Parallelization of SVD has a goal to significantly reduce its computational time on modern multi-core processors. One of the popular algorithms for SVD is the use of the power method, which helps to computes the largest singular value and corresponding singular vector. One version of parallelizing the power method would be to do matrix and vector operations in parallel. Another parallelization method involves the parallel calculation of the U matrix in SVD. Other components of the calculation process of SVD, such as normalization, matrix-vector multiplication (GEMV), and matrix-matrix multiplication (GEMM) are also good candidates that are amenable to parallelization. Various parallelization techniques (SIMD instruction level parallelism, cache optimization, multi-threading, etc) are considered in this project to attempt similar speedups from similar projects and in the existing literature. These techniques can be implemented using parallel programming models, such as OpenMP and AVX intrinsics, to efficiently exploit the parallelism available in modern multi-core processors.

1 Background and Significance

The problem we are investigating is creating a computation algorithm for Singular Value Decomposition (also known as SVD). SVD is a linear algebra-based technique that involves matrix factorization to break a large matrix A into three smaller submatrices U, Σ, and V^T: $A_{m\times n} = U_{m\times m} \cdot \Sigma_{m\times n} \cdot V^T_{n\times n}$

In the above expression, A is the original matrix that is of size $m \times n$. U is an orthogonal matrix, Σ is a diagonal matrix, and V^T is an orthogonal matrix that represents the transpose of a matrix V (which will be explained in detail later in this section). One of the most common methods of computing the singular value decomposition of a matrix $A_{m\times n}$ (for $m > n$) is through the process of eigenvalue decomposition of the symmetric matrix $A^T \cdot A$, where A^T represents the transpose of the matrix A. The corresponding singular value σ_i for each eigenvalue of the matrix product $A^T \cdot A$ is represented in the diagonal entries of the matrix Σ, where each $\sigma_i = \sqrt{\lambda_i}$. The non-diagonal entries of the matrix Σ are values of zero.

The columns of the matrix V are the eigenvectors that correspond to each of the eigenvalues of the product $A^T \cdot A$, in alignment with the singular values that are contained in the matrix Σ. The matrix V^T is simply the transpose of this matrix V and is needed in the matrix factorization of $A = U \times \Sigma \times V^T$.

$V = \begin{bmatrix} \vec{e}_1 & \vec{e}_2 & \cdots & \vec{e}_r \end{bmatrix}$, where r is the rank of the matrix A.

Finally, the matrix U is calculated as follows: $U = A \times V \times \Sigma^{-1}$

Our main research method was to delve into the mathematical model for SVD and create a standard sequential representation for each part using code. Doing so allowed us to measure runtime the performance of each part of the SVD algorithm and how they each affected the overall computational runtime.

Singular Value Decomposition is at the forefront of modern data analysis and computation. The computational algorithm has countless applications, but included in its most popular areas is image compression. In image compression, an image is originally represented as a large matrix contained several amounts of data. However, it is not always efficient to rely on large amounts of data to represent an image. SVD can be used to reduce the amount of data that is required to represent an image without sacrificing image quality too much by factoring the

matrix pertaining to the original image using singular values and eigenvectors. This technique is extremely popular all around the world. Another application is Principal Component Analysis, which relies on SVD to dissect trends across a time series within a dataset.

With how prominent SVD, it is crucial to have an efficient implementation. While a sequential SVD computation may be accurate and work well with smaller problem sizes, the computational process may not scale well when working with larger sizes of data. Indeed, performing a sequential SVD may be computationally expensive when matrices of higher dimensions are encountered. J. SairaBanu, Rajasekhara Babu and Reeta Pandey found that the SVD algorithm can be parallelized, and such an implementation was had faster performance than the sequential version, observing faster runtimes from $10 - 16\%$ across a wide range of inputs.

2 Scientific Goals and Objectives

The objective of this project is to speed up a manual implementation of SVD for use in aforementioned applications such as Principal Component Analysis and Image Compression.

The general commercial scale of these applications usually involve matrices with relatively large dimensions, such as a dataset for PCA that includes time series of decades and tens of variables or a 1024×1024 pixel square image for compression purposes. Our scientific goals are to provide a parallelized version of SVD that can be both used for these commercial-sized applications and scaled up for even larger datasets.

Considering that iterative algorithms for calculating the SVD of large matrices can become a bottleneck for these applications, our highest aim is that our parallelized version can be used for scenarios such as climate analysis involving time series of days (i.e. temperature records per day for 100 years is a time series of 36,500 points) or very high quality images (i.e. 4K resolution images that are 2,160 pixels tall and 3,840 pixels wide). For these reasons of large-scale data, the need for compute hours on a HPC architecture to run our parallel implementation of SVD is justified, especially if used by a target audience such as climate scientists and users of large-quality images.

3 Algorithms and Code Parallelization

Main algorithm

We are the main developer of this SVD algorithm. This implementation of SVD consists of the following major subalgorithms: Calculate $A^T A$; Calculate the eigenvalues and eigenvectors of $A^T A$; Construct Σ; Construct V; and Calculate and construct U. This process follow exactly from the standard SVD algorithm for a matrix with dimensions $M \times N$ ($M > N$). Each of these subalgorithms has been implemented in a major code segment that handles its operations, either in standalone control structures or in isolated methods described further below. To assist in the calculation of these subalgorithms, we have manually defined a set of relevant matrix and vector operations. These act on vectors given by the `std::vector` type and matrices given by our defined `Matrix` type (2-D vector, `std::vector(std::vector)`).

1. `gemm(Matrix, Matrix)` — General Matrix-Matrix multiplication
2. `gemv(Matrix, vector)` — General Matrix-Vector multiplication
3. `dot(vector, vector)` — Dot product between two given vectors
4. `vector_norm(vector)` — Norm of a vector
5. `normalize(vector)` — Normalize the given vector
6. `scalar_vector_mult(double, vector)` — Scalar-Vector multiplication
7. `transpose(Matrix)` — Matrix transpose
8. `matrix_subtraction(Matrix, Matrix)` — Matrix-Matrix subtraction
9. `vector_to_matrix(vector)` — Converts vector item to Matrix item

With these matrix and vector operations, we now describe the main subalgorithms in more detail:

1. Calculating A^TA is accomplished via the `AtA(Matrix)` method, which takes the `transpose()` of the given Matrix then multiplies the transpose and the original via `gemm()`.

2. Calculating the eigenvalues and eigenvectors of A^TA is accomplished with the `calculate_eigenmodes(Matrix)` method. This step requires an algorithm for computing eigenvalues and their corresponding eigenvectors. In this implementation, we use the power method as an iterative approach for calculating the dominant eigenvalue. Then, the `find_eigenvector()` method is used to find the corresponding eigenvector for the dominant eigenvalue. After this the matrix is deflated using the `deflate_matrix()` method. Once this is done, we simply return the final eigenvalues and eigenvectors.

3. Constructing the matrix Σ is done by using the `sqrt_vector()` method, which takes the square root of each entry in a given vector. In this case, the square root of all the eigenvalues are taken in order to find the singular values that will be placed along the diagonal of the matrix Σ, in descending size.

4. Constructing the matrix V is accomplished with the `transpose()` method, by taking the transpose of the matrix containing all the eigenvectors such that the eigenvectors are columns, sorted according to the sorting of the singular values in Σ.

5. Calculating and constructing the matrix U is done by the `calculate_U()` method, which uses `gemm()` for $A \cdot V$ and $(A \cdot V) \cdot \Sigma^{-1}$, `tranpose()`, `scalar_vector_mult()`, `dot()`, and `normalize()`.

The main scientific libraries that we used in our algorithm include chrono, cmath, iostream, vector, and random, most of which are for testing purposes.

Identifying Bottlenecks

After our sequential code was implemented, we measured the runtime of each of the key functions within the SVD calculation and the eigen-decomposition calculation. We measured the amount of time each function took in relation to the dimensions of the matrix that it was running calculations on.

To measure the performance of functions in the SVD calculation, we ran initial tests on matrices of up to size 250×175. When running these initial tests, our plan was to test on matrices of up to size 250×250, but the job that we submitted to the Harvard academic cluster timed out before we were able to get all our results back. Ultimately, we found that eigenmode calculation and calculation of the matrix U were the largest bottlenecks that slowed down overall performance. A plot of the performance of these two functions is shown below (note that we created plots for the other functions, but we omitted them in this project report for the sake of conciseness)

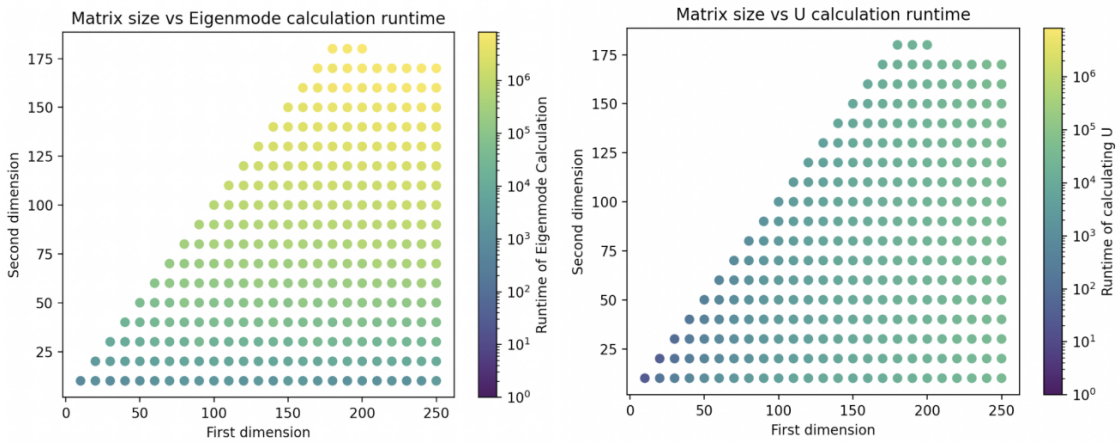

Figure 1: Visualizations of bottlenecks present in SVD calculation: Eigendecomposition of A^TA and matrix U calculation

To measure the performance of functions in the eigen-decomposition calculation, we measured runtimes of each function on matrices of up to size 250×250. After doing so, we found that the power method was much more computationally expensive in relation to the other functions. A plot of the runtime performance of this function is displayed below (again, we created plots for the other functions, but we omitted them in this project report for the sake of conciseness)

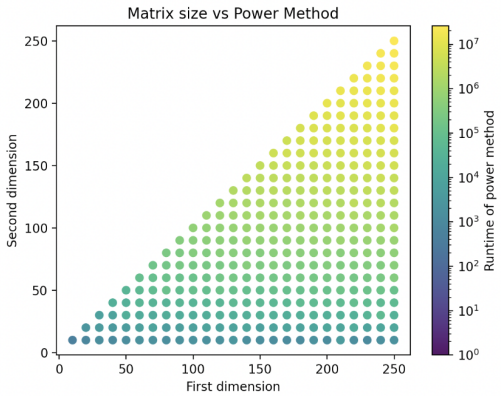

Figure 2: Visualizations of bottleneck present in the power method in the eigen-decomposition process.

Parallelization of Sequential Code

OpenMP is used in `gemv` function to parallelize the outer loop of the matrix-vector multiplication operation. This loop iterates over each row of the matrix, and each iteration is independent of the others; so, we wished to distribute the iterations across multiple threads. SIMD is also employed inside `gemv`, just like in `dot`, to perform vectorized operations on multiple data elements simultaneously to reduce the workload for each loop by taking advantage of the 256-bit wide registers, which can hold four double-precision floating-point values, while gaining on time as well. Overall, we theoretically achieve better computational efficiency by using both OpenMP and SIMD parallelization methods.

The calculations of the `normalize`, `scalar_vector_multiplication`, and `transpose` functions are dominated by the for loops inside them. Therefore, as inputs get large, the best parallelization method as expected was to take advantage of OpenMP's `#pragma omp parallel` to split the loop iterations since they are independent of each other, allowing us to do division, multiplication, and matrix traversal on separate multiple threads at a time respectively for `normalize`, `scalar_vector_multiplication`, and `transpose`. Further, for transpose, we make use of the collapse(2) clause to take care of the parallelization of the the two nested loops that traverse the rows and columns of the input matrix A together, avoiding overheads. Additionally, SIMD intrinsics are used to transpose the 4x4 block of the input matrix A. The intrinsic functions load 4 consecutive elements of each row in the block, then transpose them and store them in the output matrix B. We do this by leveraging the shuffling and permutation functions from AVX instrinsics. This approach leverages the CPU's vector processing capability and allows processing multiple elements at once, further speeding up the computation.

We also wished to parallelize the `GEMM` function. To increase optimization levels from the unoptimized version to get even faster, we wanted to combine AVX intrinsics, tiling, and OpenMP. Akoushideh et al. shows that SSE and AVX have better performance for small matrix sizes compared to OpenMP, as for matrices larger than 2048, there was no significant speedup for AVX. But for overall test benchmarks, OpenMP proves to be the most stable even though it has small benefit for small matrices, which makes sense since SIMD and AVX intrinsics are most powerful for small matrices, while the advantage of OpenMP's multiple threads gets significant with the size of the problem. AVX however is helpful in our specific problem of matrix multiplication as the ideal scenario would be to shorten the runtimes spent on floating-point operations and fetching from memory. Loop unrolling is another method that is often used to limit aggressive fetching from memory making a sort of space-time tradeoff. Other studies have also shown that cache blocking and other cache-aware methods can be pretty stable across problem

sizes, achieving speedups of up to 20 or 15. However, the best methods used tend to involve some combination of all the methods mentioned above. We tried to implement these methods fully but were limited on time.

The `power_method` depends on function calls to the `gemv` function to calculate the matrix-vector product $A \cdot q$ and on the `normalize` function is used to normalize the result, while the `dot` is also used. Therefore, from our parallelization strategies for the `gemv_intrinsics_parallel`, `dot_parallel_intrinsics`, and `normalize`, we already have much of the performance improvement. Despite the temptation of trying to use OpenMP on the iterations of the power method function, we realized that the data dependency in-between iterations make it impossible to split between threads and thus we limit ourselves to the improvement achieved in the modular functions. As for the `calculate_U` function, the `gemm` and the `transpose`, `scalar_vector_mult` functions are used and thus `calculate_U` should benefit as a result from the speedup and improvement achieved from those as well. However, unlike the `power_method` function, `calculate_U` has other unoptimized for loops which we dedicate to `#pragma omp parallel` for directives for further speedup. In fact, this is important for the bigger SVD calculation process as `calculate_U`, just like `power_method`, is central to the overall runtime and speedup and was a very significant bottleneck from our sequential runtime analysis.

Validation, Verification

We were able to validate our parallelized SVD algorithm by looking at research and experiments surrounding parallelizing matrix/vector operations. This is because we parallelized our SVD algorithm by parallelizing the helper functions that pertained to various types of vector and matrix computations. Validating our method and results of the parallelized functions with past research ensured that our approach to parallelize our SVD algorithm was accurate and efficient.

Hassan et al. found that an efficient implementation of matrix-vector multiplication resulted in a performance improvement in the range of $14-18\%$. Our results align with these results but to a much higher degree, since testing our parallelized matrix-vector multiplication resulted in an improvement of 39% (1319 ms in sequential `gemv` vs 767 `gemv` in a 256×256 matrix). Similarly, Hemeida et al. found that an efficient implementation of matrix-matrix multiplication resulted in a performance improvement in the range of approximately $50-70\%$ depending on matrix size. Again, our parallelized matrix-matrix function aligns with their result but to a slightly lower degree, as we recorded a 41% improvement (747085 ms in sequential `gemm` vs 452517 ms in parallelized `gemm` in a 256×256 matrix). Nonetheless, the aforementioned findings from researchers used specialized implementations that are expected to be more efficient than our implementation as we did not fully exploit every possible area of improvement. However, our results being aligned with the general findings is sufficient to validate our method of parallelizing the SVD algorithm, as our parallelized matrix-vector operations generate a performance improvement.

4 Performance Benchmarks and Scaling Analysis

Roofline Analysis

Ridge Point Calculation: We performed the roofline analysis for both the sequential and multi-core case on Intel Xeon E5-2683v4.
We ran our program on Intel Xeon E5-2683v4, which has the following hardware specs: base clock rate $f = 2.10 \times 10^9$cycle/s, SIMD vector width $w_s = 256$bit, $\phi = 4$ Flop/cycle with FMA3 instructions, Actual frequency $f_{\text{DDR}} = 1200$MHz, Maximum memory channels $c = 4$.

Sequential: For the sequential roofline analysis, based on the above parameters, we can calculate that the nominal peak arithmetic performance on a single core for double precision is $\pi = f \times n_c \times l_s \times \phi = 2.10 \times 10^9 \times 1 \times \frac{256}{64} \times 4 = 33.6$ GFlop/s. The nominal peak memory performance with double data rate and 64 bits through each channel per cycle is $\beta = 2 \times f_{DDR} \times c \times w = 2400 \times 4 \times 64 \times 1/8 = 76.8$ GB/s. Thus, the ridge point $I_b = \frac{\pi}{\beta} = \frac{33.6 \text{ GFlop/s}}{76.8 \text{ GB/s}} = 0.4375$ Flop/Byte.

Parallel: When we execute the parallelized program with 16 cores, the nominal peak arithmetic performance becomes $\pi = f \times n_c \times l_s \times \phi = 2.10 \times 10^9 \times 16 \times \frac{256}{64} \times 4 = 537.6$ GFlop/s. The nominal peak memory performance is still 76.8 GB/s. Thus, the ridge point $I_b = \frac{\pi}{\beta} = \frac{537.6 \text{ GFlop/s}}{76.8 \text{ GB/s}} = 7$ Flop/Byte.

Operational Intensities: Next, we calculate the operational intensities of the two main computational bottlenecks in our program – the power method and the calculation of U. We assume the size of the matrix is M and the number of iterations in the power method is k.
We first calculated the total number of flops and the total number of bytes of DRAM traffic in some important matrix operations:

- `GEMM`: $2M^3$ flops, $24M^3$ bytes (in the innermost loop, each iteration takes 1 add, 1 mul, 2 reads, 1 write).
- `GEMV`: $2M^2$ flops, $24M^2$ bytes (in the inner loop, each iteration takes 1 add and 1 mul, 2 reads and 1 write).
- `dot product`: $2M$ flops, $16M + 8$ bytes (in the inner loop, each iteration takes 1 add, 1 mul, 2 reads; 1 write at the end).
- `normalize`: $3M + 1$ flops, $32M + 8$ bytes (calculating the vector norm takes $2M + 1$ flops and $16M + 8$ byte; afterwards, 1 flop, 1 reads, and 1 for each of the M elements in the vector).

Power Method: In each iteration, in terms of the compute performance, `normalize` takes $3M + 1$ flops, `gemv` takes $2M^2$ flops, `dot` takes $2M$ flops, and $\lambda_0 \leftarrow \lambda_1$ takes 1 flop, so the total number of flops per iteration is $2M^2 + 5M + 2$.
In terms of the memory performance, `normalize` takes $32M + 8$ bytes, `gemv` takes $24M^2$ bytes, and `dot` takes $16M + 8$ bytes, so the total number of bytes of DRAM traffic is $24M^2 + 48M + 16$.
We see that `gemv` accounts for both the main compute cost and the main memory cost.
The operational intensity is thus $\dfrac{2M^2 + 5M + 2}{24M^2 + 48M + 16} \approx 0.083$ Flop/Byte.

Calculation of U: If the matrix is square or tall and thin, the calculation of Σ_{inv} takes M flops and $16M$ bytes, and the two gemm functions each takes $2M^3$ flops and $24M^3$ bytes, so in total $4M^3 + M$ flops and $48M^3 + 16M$ bytes. The operational intensity is thus $\dfrac{4M^3 + M}{48M^3 + 16M} \approx 0.083$ Flop/Byte.

If the matrix has the number of columns $M >$ the number of rows, we need the Gram Schmidt. Inside the inner loop, each dot product takes $2M$ flops and $16M + 8$ bytes; each scalar vector multiplication takes M flops and $16M$ bytes; each vector subtraction takes M flops and $24M$ bytes. In the outer loop, the normalization takes $3M + 1$ flops and $32M + 8$ bytes. Each transpose takes $16M^2$ bytes. In total, there are additionally $4M^3 + 3M^2$ flops and $56M^3 + 64M^2$ bytes, so the total number of flops is $8M^3 + 3M^2 + M$ and the total bytes transferred is $104M^3 + 64M^2 + 16M$. Thus, the operational intensity is $\dfrac{8M^3 + 3M^2 + M}{104M^3 + 64M^2 + 16M} \approx 0.076$ Flop/Byte.

With these values and assuming the matrix is wide, we created the roofline plots below for both single core and 16 cores. We see that the operational intensities of both the power method and the calculation of U are on the left side of the ridge point, so they are both memory-bound. As we increase the number of cores, the ridge point shifts to the left, which makes the operational intensities even further away from the ridge point. This analysis suggests that both functions might not scale well with the increasing number of cores.

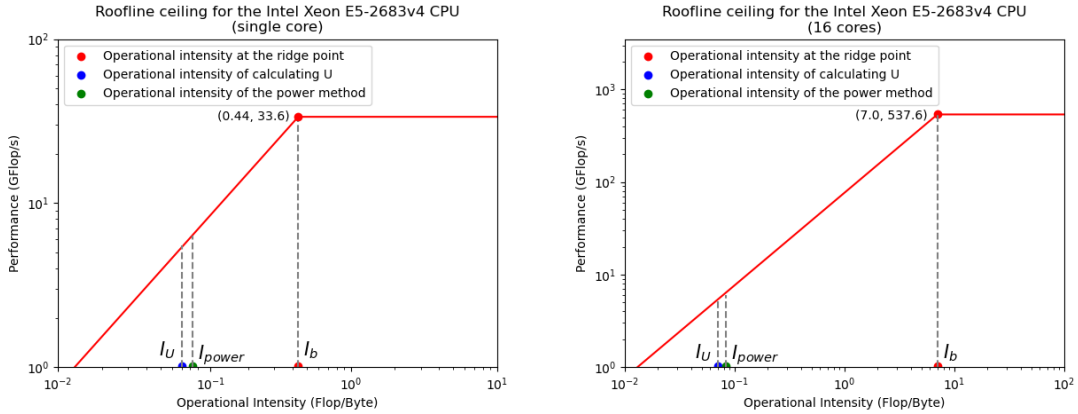

Figure 3: Roofline Ceiling plots for single core and 16 cores on the Broadwell Node on the Cluster.

Strong Scaling Analysis: 256×256 and 512×512 matrices using parallelized SVD algorithm

Since we implemented a shared memory program with relatively few processors and negligible communication overhead, to examine the efficiency and scalability of our implementation, we performed a strong scaling analysis on the overall performance, the power method, and the calculation of U. In the strong scaling analysis, we fix the problem size and hope to observe the scaling effects in the execution time as we increase the number of processors. The table below displays the total runtime, the power method runtime, and the `calculate_U()` runtime (measured in microseconds) for the sequential baseline and the parallelized program with different core counts, ranging from 1 to 16 cores.

Number of cores	Total Runtime	Power Method Runtime	Calculate U Method Runtime
Sequential Baseline	2.37765e+06	2.11713e+06	58360
1	2.45836e+06	1.16565e+06	36947
2	6.78526e+06	2.13537e+06	36257
4	8.34796e+06	2.07544e+06	36571
8	1.15366e+07	2.1885e+06	36020
16	1.77818e+07	2.27983e+06	33411

Table 1: Runtime (microseconds) of the sequential baseline and our optimized implementation (with OpenMP, tiling, AVX instructions, and other cache-aware methods) with respect to the number of cores. The matrix size is 256×256.

The results show some level of speedup when we compare the runtime of the sequential baseline with the runtime of our optimized implementation (with tiling, AVX instructions, and other cache-aware methods) running on a single core. However, when we increase the number of cores from 1 to 16, our implementation shows speedup only for our `calculate_U()` function. The power method and thus the overall performance (since the power method dominates the overall runtime) both experience increasing runtime with the increase in the number of cores. This might due to the huge parallel overhead caused by the power method calling multiple parallelized functions (GEMV, normalize, dot, etc.) at every iteration and requiring thread synchronization each time, or since there may be more cache misses with more cores.

The data also suggests that for our `calculate_U()` function, the runtime does not decrease proportionally with the increase in the number of cores, indicating that the algorithm may not be achieving optimal strong scaling. However, there is still a noticeable improvement in performance when comparing the runtime of a single core to that of 16 cores.

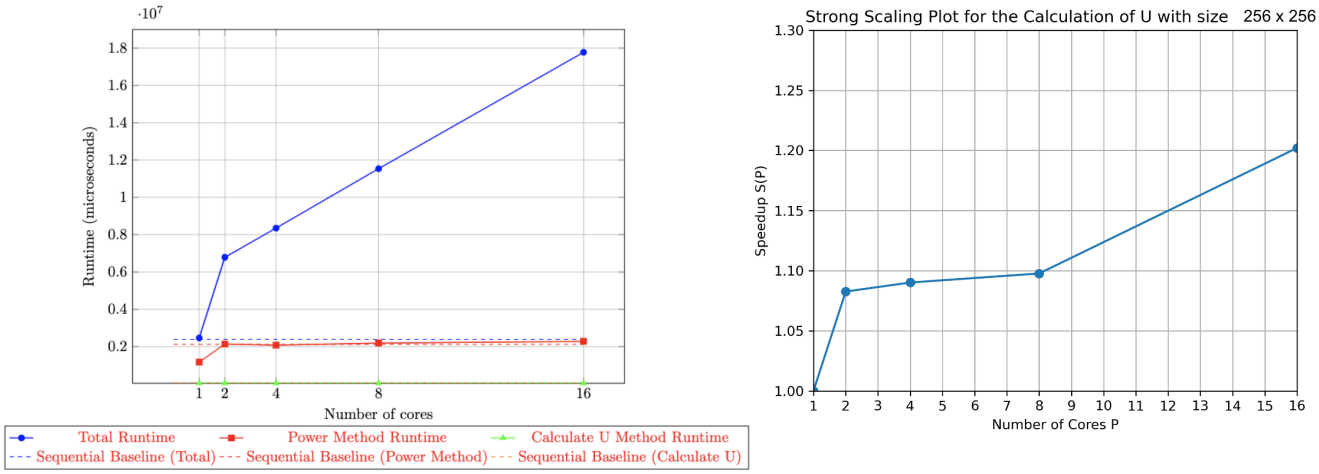

Figure 4: Plot on left side visualizes the data shown in table 1. Note that the Calculate U runtime is hard to visualize as the order of magnitudes of the data points are much smaller than the other data points. We make the analysis of Calculate U more clear using the plot on the right, which shows the scale up of the Calculate U function as the number of cores increases. The matrix size is 256×256.

To investigate the worsening in the runtime of the optimized implementation of the power method, we tested the performance of an implementation with only the for loop in the GEMV function being parallelized trivially using OpenMP. As a result, we observed a slight but steady decrease in the runtime as we increase the number of cores. This suggests that our implementation might suffer from the increasing cache misses, the unnecessary SIMD/AVX operations on vectors of relatively small dimensions, and the adverse interference between the SIMD and OpenMP implementations. Thus, further analysis needs to be conducted to see if a better speedup can exist.

Number of cores	Total Runtime	Power Method Runtime
Sequential Baseline	1.8419e+07	1.66112e+07
1	1.80933e+07	1.66147e+07
2	1.78484e+07	1.62603e+07
4	1.58207e+07	1.42126e+07
8	1.50523e+07	1.33553e+07
16	1.49246e+07	1.31571e+07

Table 2: Runtime (microseconds) of the sequential baseline and an implementation with only GEMV trivially parallelized using OpenMP. The matrix size is 512×512.

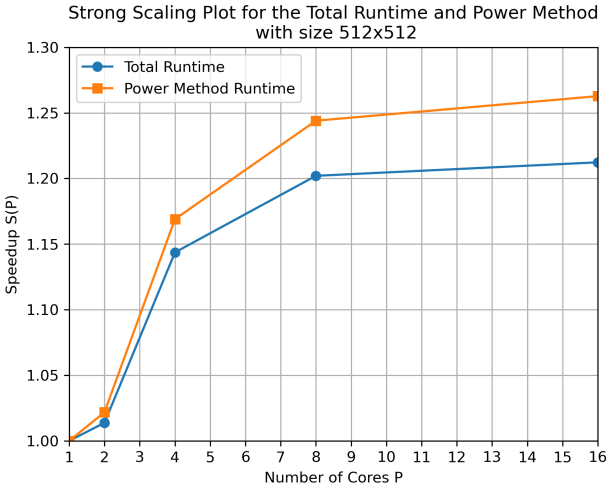

Figure 5: The strong scaling plot for the total and the power method runtime when only GEMV is trivially parallelized with OpenMP. The matrix size is 512×512.

	Test case A	Test case B
Typical wall clock time (hours)	0.1400206	0.1121053
Typical job size (cores)	16	8
Memory per node (GB)	8	8
Maximum number of input files in a job	0	0
Maximum number of output files in a job	0	0
Library used for I/O	n/a	n/a

Table 3: Workflow parameters of two test cases (16 cores vs 8 cores, 256×256 matrix).

5 Resource Justification

In order to carry out the SVD calculation on large-scale matrices, significant computational resources are required. Several sub-operations of the SVD calculation are required to be parallelized, such as power method, calculate_U, normalize, dot, gem, transpose, scalar_vector_mult, etc. The number of iterations required for each sub-operation depends on the size of the matrix and the desired accuracy of the result. The number of iterations of the power method functions bounds the wall time. To estimate the total resource requirement, we considered a typical test case with matrices of size M and N = 256 or 512. We make the following assumptions: will use at most 16 cores for parallel processing (2 cores, 4 cores, 8 cores, 16 cores) and we will use additional test cases specifically for the major parts of the project (for instance, Power method). Our biggest test size of 512x512 for 16 cores runs in a wall time of $1.49246e7$ microseconds, which is then equivalent to:

$$0.0663 \text{ core hours} = 16 \text{ cores} \times \frac{14.9246s}{3600\frac{s}{\text{hour}}}$$

For the application of image compression using SVD, using an original image of size 512x512 and retaining 16 singular values, thus achieving a compression ratio of 32:1, would require 5 iterations of the SVD calculation since each iteration of SVD reduces the number of singular values by 2. Such a high level of compression may result in some loss of detail and information in the image, but the resulting image would still be recognizable and usable for many applications. If we started with a batch of about 100 images, which would be a good start to see how well our compression works on a sizeable sample, we would end up needing about 500 total iterations of the SVD calculation.

For the similarly important application of PCA, with a dataset with 512 observations and 32 features and a PCA analysis that attempts for the best possible total explained variance in the data and good enough approximations of the singular values and vectors, we would need about 10 to 20 iterations in practice. If we assume to run PCA on a stream of separate data divided into 50 samples for instance (since the size of observations tends to be large and validation methods tend to rely on down-sampled data that are run separately), we can expect about 1000 total iterations of the SVD algorithm to achieve good PCA analysis of our entire dataset.

We would also wish to run multiple simulations to validate our runtimes. Often for image compression, not many simulations would be needed (so about 3 would be a good bound), while for PCA, we might want to verify our approximations and explained variance pretty closely, so a bound of about 5 would do well in this instance.

	Test case A	Test case B
Simulations per task	3	5
Iterations per simulation	500	1000
Code hours per iteration (upper-bound)	0.0663	0.0663
Total core hours	99.45	331.5

Table 4: Justification of the resource request

6 Discussion and Future Considerations

Overall, we were able to see that parallelizing the vector/matrix operations of the SVD algorithm resulted in speedups of those respective parts. However, we note that our approach has issues and limitations as our results did not align well with our expectations. This can be seen in our runtime results as the parallelized vector/matrix operations slowed the overall SVD runtime. If we had more time, we would like to investigate if the issue in our parallelized code was caused by our use of OpenMP or the use of intrinsics. Since our small test case with trivial OpenMP directives resulted in a (very slight) speedup, we can assume that our managing of SIMD AVX instructions may have resulted in either poor cache/memory use, increased overhead, or subtle accumulation of unnecessary work across threads which may have caused the execution time to increase with the number of cores.

Additionally, our investigation found that the increase in instructions in general (i.e. for SIMD/AVX) caused a slowdown for higher thread counts, and that the power method had a very imbalanced number of iterations

for certain eigenvalues (our example matrices's eigenvalues ranged from tens of iterations to up to 7000 iterations before convergence). If we further explored this project, we would definitely attempt to bound the number of iterations for the power method to reach approximate solutions instead of fuller convergence, which in turn would help simplify the computational intensity and analysis of the power method. This would also reduce the power method's contribution to runtime significantly and move it away from its role as a bottleneck.

The current comparison is based on both the sequential baseline and the parallelized implementation complied with the -O3 flag. Another thing that we would like to try is to make a comparison without the -O3 flag for both the sequential and the parallel program. Although this would make our computations much slower, a high optimization level might lead to unpredictable performance improvements because it could potentially optimize away many of our implementation designs and some optimization techniques such as compiler auto-vectorization and compiler-generated parallelism might defeat the speedup provided by our SIMD/OpenMP implementation. It may likely be the case that the compiler optimization from this flag results in specially quick code in the sequential case that does not translate to our more complex implementations (e.g. compiler loop unrollings for basic loops not carrying over to our specially designed AVX loops). Further, aggressive optimizations might interfere with the parallel code by introducing race conditions or other synchronization issues, leading to unexpected behavior or even incorrect results.

Besides these points, it is important to note that we aimed to parallelize a more popular and more standard algorithm for SVD using eigendecomposition, which is more widely accessible and easier to grasp. Literature suggests that there are other specially designed algorithms for SVD that may have more affinity to parallelization, such as repeated matrix column orthogonalizations, but these would be more difficult to follow. Thus we chose to parallelize a more widely known SVD algorithm instead.

References

[1] Akoushideh, A., & Shahbahrami, A. (2022, October 11). Performance Evaluation of Matrix-Matrix Multiplication using Parallel Programming Models on CPU Platforms (Version 1) [Preprint]. Research Square. https://doi.org/10.21203/rs.3.rs-2135830/v1

[2] Hassan, S., Mahmoud, M., Hemeida, A., Saber, M. (2018, January). Effective Implementation of Matrix–Vector Multiplication on Intel's AVX multicore Processor. Computer Languages, Systems & Structures. https://doi.org/10.1016/j.cl.2017.06.003

[3] Hemeida, A., Hassan, S., Alkhalaf, S., Mahmoud, M., Saber, M., Elden, A., Senjyu, T., Alayed, A. (2020, December). Optimizing matrix-matrix multiplication on intel's advanced vector extensions multicore processor. https://doi.org/10.1016/j.asej.2020.01.003

[4] SairaBanu, J., Babu, R., Pandey, R. (2015, July). Parallel Implementation of Singular Value Decomposition (SVD) in Image Compression using Open Mp and Sparse Matrix Representation. Indian Journal of Science and Technology. https://sciresol.s3.us-east-2.amazonaws.com/IJST/Articles/2015/Issue-13/Article8.pdf