An Investigation into Parallel SVD for MATH-GA.2011.001: Andreas Kloeckner and Marsha Berger

Travis Askham, Steven Delong, and Michael Lewis
Courant Institute of Mathematical Sciences

December 23, 2012

Abstract

We discuss an investigation into parallelizing the computation of a singular value decomposition (SVD). We break the process into three steps: bidiagonalization, computation of the singular values, and computation of the singular vectors. We discuss the algorithms, parallelism, implementation, and performance of each of these three steps. The original goal was to accomplish all three tasks using a graphics processing unit (GPU) but the final implementation uses a combination of GPU computing and multicore central processing unit (CPU) computing. The two parallelization standards used were OpenCL and OpenMP. Our SVD implementation and its components are available freely online.

1 Introduction

The computation of an SVD is ubiquitous in numerical computing and has many applications [1], [2], [3]. Given an $m \times n$ matrix A, its SVD is

$$A = U\Sigma V^*$$

where $U \in \mathbb{R}^{m\times m}$ and $V \in R^{n\times n}$ are orthogonal matrices and $\Sigma \in R^{m\times n}$ is diagonal with nonnegative entries. The SVD allows for optimal low rank approximations to the original matrix and the formation of a pseudo-inverse. It has applications in diverse areas, including data compression and signal processing.

We investigate the use of parallelization methods to rapidly compute an SVD. Our algorithm is broken into three stages: a bidiagonalization step, computation of the singular values, and computation of the singular vectors. The bidiagonalization step is the most expensive and is carried out purely on the GPU, using the OpenCL standard; we describe this step in Section 2. We follow a "Double Divide and Conquer" approach as described in [4] for the following two steps and we detail our findings in Sections 3 and 4. The parallelization of these steps is achieved primarily through multithreaded computing using the OpenMP standard.

Our target problem size is determined by the bidiagonalization step. We consider matrices whose side lengths are approximately 1,000 to 10,000. The lower end of this range is determined by the overhead of transferring data to the GPU and the upper end of the range is determined by the amount of data that can be stored in GPU memory.

2 The Bidiagonalization Step

The standard approach to computing an SVD first reduces the matrix to bidiagonal form [1]. In particular, for a given matrix A, an upper-bidiagonal matrix B and two orthogonal matrices U and V are found such that

$$A = UBV^* \tag{1}$$

Then, the SVD of the bidiagonal matrix B is found and combined with (1) to produce an SVD for A. This inititial reduction to bidiagonal form allows for the calculations in the remaining steps to be sparse. It is for this reason that the bidiagonalization step is the most expensive computationally.

2.1 The Algorithm

We use the standard Golub-Kahan [3] bidiagonalization algorithm. This algorithm proceeds by introducing zeros below the main diagonal and above the super-diagonal via Householder reflections. Each step of the algorithm reduces the problem to that of bidiagonalizing the matrix which remains after removing the first column and first row of the starting matrix. We describe one such step below. Let A_k be your starting matrix. We would like to find Householder reflectors U_k and V_k such that

$$U_k A_k V_k^* = \left(\begin{array}{c|c} \alpha_k & \beta_k \\ \hline & \\ & A_{k+1} \\ & \end{array}\right) \tag{2}$$

Let x be the first column of A_k. To find (2), we must calculate a reflector which will zero out the entries below the first entry in x. A stable [1] choice for this reflection vector is $v = \text{sign}(x_1)\|x\|e_1 + x$. Setting $U_k = I - 2vv^*/(v^*v)$ will result in

$$A_k = \begin{pmatrix} \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \\ \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} \end{pmatrix} \to U_k A_k = \begin{pmatrix} \alpha_k & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \end{pmatrix} \tag{3}$$

Similarly, if we let r be the partial row starting at $(U_k A_K)_{(1,2)}$ we can find a reflector which – when applied on the right – zeros out the first row of $U_k A_k$ to the right of $(U_k A_K)_{(1,2)}$. If we let w be this reflector, then setting $V_k = I - 2ww^*/(w^*w)$ will result in

$$U_k A_k = \begin{pmatrix} \alpha_k & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \\ 0 & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} & \tilde{x} \end{pmatrix} \rightarrow U_k A_k V_k^* = \left(\begin{array}{c|c} \alpha_k & \beta_k \\ \hline & \\ & A_{k+1} \\ & \end{array}\right) \tag{4}$$

The process is then applied to A_{k+1}. Thus, we see that at each step we must calculate a left reflection vector, apply a reflection on the left, calculate a right reflection vector, and apply a reflection on the right.

2.2 Parallelism

The algorithm described above is rich in parallelism and it is reasonably straightforward to parallelize on a GPU [2]. There are two main types of tasks for this algorithm. The first is to calculate a reflection vector and the second is to apply that vector to the matrix.

The main requirement in calculating a reflection vector is finding the norm of a partial column or partial row, denoted by x. As in [2], we perform this calculation using a reduction. Specifically, we calculate the norm squared of x via a reduction and store the result in global memory. Each work item in the initial step of this reduction stores a chunk of x in local memory and calculates the sum of squares for that chunk. These chunks are then further reduced using a standard sum redution algorithm. Once we have calculated the norm of x, the reflection vector can be stored in place in the original matrix A by simply modifying the first entry of x and then scaling. The scaling of each element in the modified x is independent of the other elements. However, there is not much to be gained here as this step is light in terms of computation and heavy in terms of global memory access.

Once the reflector is calculated, it is applied to a submatrix C of our original matrix A. We divide this into two steps. First, we must calculate the inner product of the reflection vector with the columns (left reflection) or rows (right reflection) of C. The inner product calculation for each column or row is independent of the others. We limit our description to the case of a left reflection for the sake of concreteness. Let v be the normalized reflector to be applied to C, i.e., we would like to calculate $p = v^*C$, a row whose entries are the innerproducts of v with the columns of C. To accomplish this, we use a row of 2D workgroups which marches down the columns of C as in Figure 1. Each work item loads the corresponding data from C and then one work item per column computes the inner product for that chunk of the column with the corresponding chunk of v.

After $p = v^*C$ is calculated, the second step of applying the reflector to C modifies each entry in C. In particular, $C_{ij} \leftarrow C_{ij} - 2v_i p_j$. This step is independent for each entry in C and is limited by the necessary global memory access. We accomplish this by assigning each work item an element

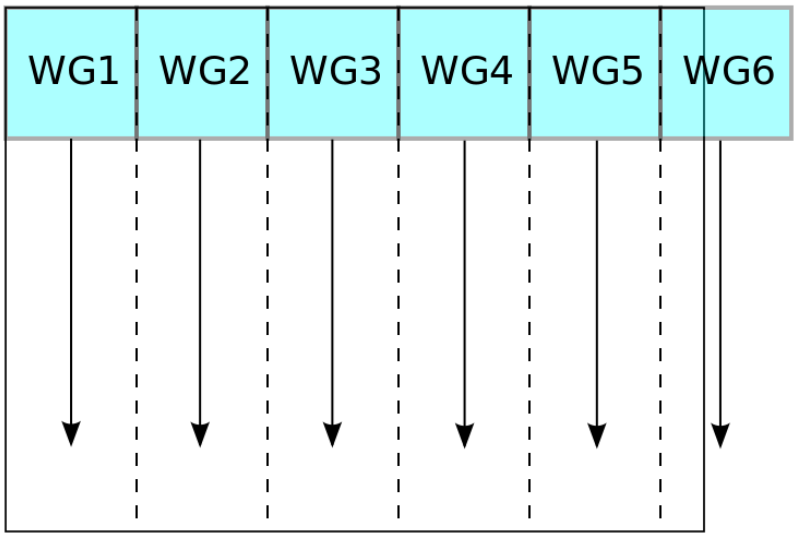


Figure 1: Work group layout for inner products

in C to modify and we lay out 2D work groups in a 2D array. The goal behind this layout is to make efficient use of global memory access. The primary observation is that elements in the same row of C are modified using the same entry of v, whereas elements in the same column of C are modified using the same entry of p.

2.3 Some Further Implementation Details

The operations described in the subsection above were implemented in the heterogeneous platform framework OpenCL. In order to do so, we first coded up a straightforward version of Golub-Kahan in C, with no parallelization. This serial version proved helpful for prototyping and is provided with the code release as described in the Conclusion. We then replaced the operations of the serial Golub-Kahan implementation with OpenCL computations in order to parallelize.

A rather naïve first approach to the calculation of the dot products v^*C was to launch a separate kernel per column and calculate the inner product via a reduction operation. This proved rather slow and we saw a significant improvement when we switched to a single kernel launch with work groups layed out as in Figure 1. Similarly, our initial approach to updating C, once v^*C was computed, was to launch a separate kernel per column of C. When we replaced this approach with the simple entry-wise approach described in the previous subsection, we again saw significant improvements in the performance of our algorithm.

We note that as the Golub-Kahan algorithm proceeds, the size of the working set becomes smaller and smaller. This creates an increasing load imbalance and an ever poorer communication to computation ratio as the calculation proceeds. There are ways to deal with this [2], but we had to ignore this issue due to time constraints. There are also alternatives to the Golub-Kahan algorithm

which are more efficient [1] in the case of tall-skinny matrices, i.e. $A \in \mathbb{R}^{m\times n}$ with $m \gg n$. In short, these alternatives choose a point in the original algorithm to perform a QR factorization on the remaining matrix, decreasing the size of the working set. We were unable to explore the effect of making such a change to the algorithm though we think it poses an interesting possibility – in particular, the motivations to decrease the size of the working set are muddled when you're computing on a GPU.

Finally, we point out that it may appear we have to calculate the norm of two vectors in order to calculate a reflection vector, one while finding the appropriate direction and the other while normalizing that direction. For concreteness, let $x \in \mathbb{R}^{n-k+1}$ be the first column of the working set A_k. The reflection vector is then in the direction $v = \text{sign}(x_1)\|x\|e_1 + x$. Thus, we must calculate $\|x\|$. Then, to apply v, we must first normalize it. However, we observe

$$\|v\|^2 = x_1^2\|x\|^2 + 2x_1^2\|x\| + x_1^2 + x_2^2 + \cdots + x_{n-k+1}^2 = x_1^2\|x\|^2 + 2x_1^2\|x\| + \|x\|^2 \tag{5}$$

so that the norm of v can be found using a slight modification to the norm of x. This is reflected in our OpenCL implementation.

2.4 Performance

While our algorithm is far from optimized, we would like to give a sense of its performance. To do so, we timed our algorithm on a range of matrices in the dimensions we were targeting and reported (see Table 1) the results in GFlops/s. The time used includes the transfer time to and from the GPU. The flop count is based on the flop count of the Golub-Kahan algorithm, $4mn^2 - 4/3n^3$ for an $m \times n$ matrix. The compute device used was an AMD Radeon Cape Verde XT (7770 HD) with 1 GB of memory and a peak theoretical performance of 80 GFlops/s.

Table 1: Bidiagonalization Performance in GFlops/s

Size	$1k \times 1k$	$2k \times 2k$	$4k \times 4k$	$8k \times 8k$
GFlops/s (GPU)	.597487	2.00485	3.3419	3.5486

While these numbers are intentionally conservative and a vast improvement over the performance of our serial implementation, they are well off the peak theoretical performance of the chip and – not surprisingly – the reported performance of other GPU implementations (for instance [2]).

3 Finding the Singular Values

In the first stage of evaluating the SVD of a general matrix, we reduced our matrix to a bidiagonal matrix. In this second stage, we are concerned with finding the singular values of the resulting bidiagonal matrix B.

3.1 The Algorithm

The details of the overarching algorithm for this section, referred to as Double Divide and Conquer (dDC), can be found in [4]. Specifically, for our purposes we assume the matrix B is $\mathbb{R}^{N\times N+1}$, with the bidiagonal elements denoted by two real vectors b_1 and b_2, each of which are of length N.

$$B = \begin{pmatrix} b_{11} & b_{21} & 0 & \dots & 0 \\ 0 & b_{12} & b_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & b_{1N} & b_{2N} \end{pmatrix}$$

This algorithm is recursive in nature, in that it solves two smaller problems before returning to solve the problem at hand. In particular, we consider the $K \times K+1$ bidiagonal submatrix B_1 and the $N-K-1 \times N-K$ bidiagonal submatrix B_2, where B_1 comes from the first K rows of B and B_2 from the last $N-K-1$ rows, resulting in

$$B = \begin{pmatrix} B_1 & 0 \\ b_{1,K+1}e_{K+1} & b_{2,K+1}e_1 \\ 0 & B_2 \end{pmatrix},$$

where e_j denotes the vector with all zeros except a 1 at index j. For the purposes of our implementation, we choose $K = \lfloor \frac{N}{2} \rfloor$. Let us assume we knew the correct SVD for the submatrices, i.e. $B_i = U_i \, (\Sigma_i \; 0) \, (V_i \; v_i)$. Then it can be shown that

$$B = \tilde{U} \, (M \; 0) \left(\tilde{V} \; \tilde{v} \right)^T \text{ where}$$

$$M = \begin{pmatrix} r_0 & b_{1,K+1}l_1 & b_{2,K+1}f_2 \\ 0 & \Sigma_1 & 0 \\ 0 & 0 & \Sigma_2 \end{pmatrix}$$

for l_1 equal to the last row in V_1 and f_2 equal to the first row in V_2. Proceeding forward, if we further decompose $M = U_M \, (\Sigma \; 0) \, V_M$ into its SVD, we find that our original matrix B can be reduced to

$$B = \tilde{U}U_M \, (\Sigma \; 0) \left(\tilde{V}V_M \; \tilde{v} \right)^T = U \, (\Sigma \; 0) \, (V \; v)^T .$$

From this formulation, we immediately see that the singular values of our matrix B are equivalent to the singular values of this special matrix M. Furthermore, letting $\tilde{f}$ and $\tilde{l}$ be the first and last rows, respectively, of $\tilde{V}$, we observe that f and l, the first and last rows of V, can be derived from the equations $\tilde{f}V_M$ and $\tilde{l}V_M$, respectively, each of which require $\mathcal{O}(N^2)$ computation. Consequently, by solving for the singular values, and first and last rows of submatrices (along with few other elements; again, see [4] for specifics) we can recursively back out the singular values for our bidiagonal matrix, all while using no more than $\mathcal{O}(N)$ space.

As for obtaining the singular values σ_i of M, it can be shown that they satisfy what is known as the secular equation, namely

$$f(\sigma_i) = 1 + \sum_j \frac{z_j^2}{d_j^2 - \sigma_i^2} = 0,$$

where z is the top row of M, and the d_j are the diagonal elements of M (where $d_1 \equiv 0$), or equivalently the singular values of the lower levels. We immediately observe, assuming the d_j are ordered, that

$$0 < \sigma_1 < d_2 < \sigma_2 < \ldots < d_N < \sigma_N.$$

Solving for these σ_i is a root finding problem, made somewhat difficult but the functional form. However, when formulated in a slightly different way (see [5]), it can be solved relatively quickly with Newton's Method. In doing so, we observe that evaluation of f at each step requires $\mathcal{O}(N)$ computation; when done over N singular values, and assuming $\mathcal{O}(1)$ iterations, we observe this section requires $\mathcal{O}(N^2)$ computation time.

3.2 Speedup Through Parallelization

The reason we originally selected this algorithm was due to its

- low memory requirements, and
- recursive nature,

both of which made it an ideal candidate for parallelization via the GPU. However, upon building the sequential code, it became apparent that the GPU wouldn't be appropriate for the recursive steps. Specifically, the step that solves the secular equation requires a variety of case handling to enhance its robustness in finding the singular values. Consequently, this left the only possibility of parallelization via the GPU to the leaf cases, i.e. the base cases that need to be solved exactly before building up the solution recursively. To that end, we have built a version of the singular value solver that does exactly that on the GPU, before building up the final result recursively on the CPU. However, for the purposes of testing, we also created a version of the singular value solver that computes exclusively on the CPU.

Independent of the GPU, we notice that the recursive steps, namely the secular equation solver and the first and last line computation steps, are both embarrasingly parallel, and thus warranted use of OpenMP to speed up those steps. In particular, the secular equation solver requires solving for N different singular values, and given the above mentioned separated nature of the values, can be solved for in parallel with ease. As for the first and last line computations, these are matrix vector multiplications, and are thus also embarrasingly parallel. Both of these sections, which we observe to be $\mathcal{O}(N^2)$ computation, were thus enhanced with OpenMP parallelization.

3.3 Results of Testing

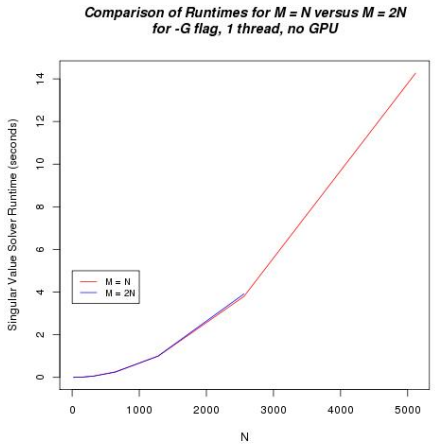
As mentioned above, two parallelized versions of the singular value solver were created, namely

(1) one where the leaves are solved via GPU, while the recursive steps are enhanced with OpenMP implementations, and

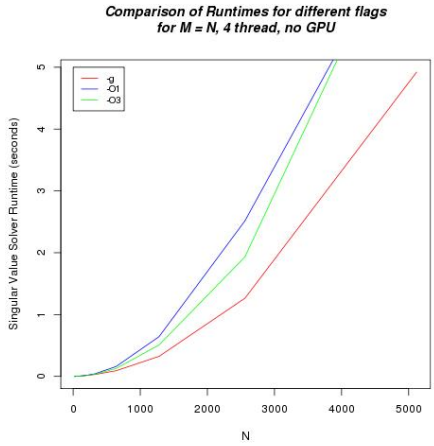
(2) one where both the leaves and the recursive steps are solved on the CPU, again with OpenMP implementations for speedup.

For simplicity, we refer to version (1) as the one that uses GPU, while version (2) is the one that does not. During testing, we ran calculations on $M \times N$ matrices for $M = N$ and $M = 2N$ and for $N = 20, 40, 80, \ldots, 5120$. Furthermore, the timing driver utilized various compilation flags (-g, -O1, -O3), and were run using various thread counts for the OpenMP (1, 2, 4, 8). The testing below was performed on a computer with a AMD FX-8150 Processor, 16 Gigabytes of RAM, with Radeon HD 7770 GPU with 1G of memory.

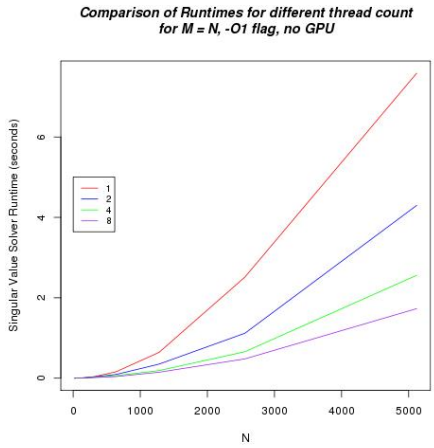
As expected, there was no discernable difference in runtime between $M = N$ and $M = 2N$, nor should we as both cases are handled the same way insofar as the singular value calculation is concerned, and thus for simplicity we assume $M = N$.



Similarly, as one might expect, compiling the results with the nonoptimized -g flag had noticeably slower runtimes than those with the other optimized flags. For example, the singular value solver without GPU for the 5120×5120 matrix took 4.92 seconds using 4 OpenMP threads under the -g flag, versus 2.55 seconds and 2.49 seconds using the -O1 and -O3 flags, respectively, resulting in a reduction of $\approx 50\%$ in runtime by optimizing in compiling. Similar speedups can be observed across different N, threads, and also with the GPU, although the speedup is less potent for smaller matrices. However, as apparent in the above example, there is little difference in runtime between using the -O1 and -O3 flag across the different dimensions; consequently, we concern ourselves only with the optimization flag set to -O1.

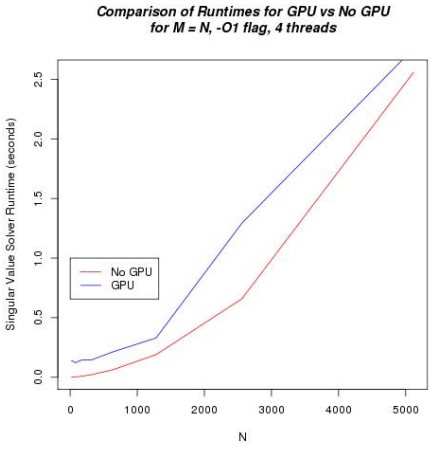


In timing our process across different numbers of OpenMP threads, there is a marked improvement as the number of threads increases, regardless of whether the GPU is used. For example, the singular value solver without GPU for the 5120×5120 matrix took 7.58 seconds using 1 OpenMP thread, 4.29 seconds with 2 threads, 2.56 seconds using 4 threads, and 1.73 seconds using 8 threads. This implies decreases in runtime of 43%, 66%, and 77% for 2, 4, and 8 threads, respectively. Similarly, in using the singular value solver with GPU for the 5120×5120 matrix took 8.05 seconds using 1 OpenMP thread, 4.55 seconds with 2 threads, 2.75 seconds using 4 threads, and 1.94 seconds using 8 threads. Again, this implies decreases in runtime of 43%, 64%, and 74% for 2, 4, and 8 threads, respectively, when compared to the runtime for one thread. These speedup improvements are observed both with and without GPU across all N, although more apparent for larger N than smaller N, as there is a sunk cost for implementing OpenMP which impacts performance at lower matrix sizes ($N < 100$).



With regards to the usage of GPU, we observe that, insofar as calculating singular values is concerned, the GPU does not add value to the computation, at least for the matrix sizes in this analysis

(namely $N \leq 5120$). In particular, there is the expected sunk cost of moving the data over to the GPU and back, and thus adding considerable time for small matrices. For larger matrices, the results are more comparable; for example the runtime on 4 threads for a 5120×5120 matrix with GPU is 2.75 seconds versus 2.55 seconds without, resulting in a 7% slowdown. The runtime on a single thread for a 5120×5120 matrix with GPU is 8.05 seconds versus 7.58 seconds without, resulting in a 6% slowdown. In light of this, running singular value evaluation in the current setup using the given algorithm is not recommended for matrices smaller than 5000 by 5000. Consequently, we concern ourselves only with the version that does not make use of the GPU.



In regards to flop calculations, we first consider the first and last row calculation. Specifically, this section is amenable to flop calculation as each for loop duration is known with reasonable accuracy, although not perfect accuracy as there are some edge cases to concern ourself with. In particular, this will be an issue as N gets larger, as the edge case is hit with higher frequency, but for this calculation we will assume no edge cases. With this in mind we observe that, for a matrix of size 2560×2560, we observe 0.122 Gflops, 0.225 GFlops, and 0.421 Glops for 1, 2, and 4 threads, respectively. This results in speedups of 83% and 242% for 2 and 4 threads, respectively, over the base case of 1 thread. These speedups are similarly observed for N down to around 320, below which the speedups are less favorable. For example, on a 40×40 matrix, we observe 0.025 GFlops, 0.030 GFlops, and 0.035 GFlops for 1, 2, and 4 threads, respectively. This results in speedups of only 16% and 35% for 2 and 4 threads, respectively, over the base case of 1 thread. One possible reason for the low level of GFlops in these results is due to the fact that, given the recursive nature of the algorithm, we are constantly moving in and out of the first and last line function call, resulting in overhead in processing the for loop.

As for the secular equation solver, due to the iterative nature of this solver, it is less amenable to flop calculation. Specifically, if the singular values are close together (as they are expected to be at the higher levels of the algorithm), there are more edge cases that need to be handled. In the case of the solver as currently implemented, if the Newton's Method is taking too long, the algorithm reverts to solving the value by the bisection method, which is notably slower. Furthemore, at the

higher levels, evaluating f will also take longer, for obvious reasons. With these caveats in mind, we can consider the rudimentary calculation of "singular values evaluated per second". We notice that, at the highest level of solving for a 500×500 matrix, we obtain only 14955 singular values per second with 1 thread versus 38194 singular values per second with 4 threads, resulting in a roughly 4 times speedup. However, at the lower levels, for example when $N = 30$, we obtain 223000 versus 917000 singular values per second with 4 threads, again resulting in a roughly 4 times speedup. From this, we see that solving the secular equation is embarrasingly parallel. Furthermore, we see that the expediency of the solver is much lower at large N when the evaluation of f takes considerably longer and the singular values are more densely situated.

4 Finding Singular Vectors

This section addresses calculating the singular vectors given singular values. First the Twisted Factorization algorithm is used to get singular vectors of the bidiagonal matrix. Then the Householder reflectors are applied to get the singular vectors of the original matrix. The algorithm used is briefly outlined below, followed by a discussion of performance.

4.1 The Algorithm: Twisted Factorization

We are given a bidiagonal matrix B

$$\boldsymbol{B} = \begin{pmatrix} a_1 & b_1 & 0 & \dots & 0 \\ 0 & a_2 & b2 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & a_n & b_n \end{pmatrix}$$

as well as its singular values σ_i.

Calculating the singular vectors of our bidiagonal matrix has the following steps, following [4], for each σ

1. Create the double factorization for each σ_i, $\boldsymbol{B}^t\boldsymbol{B} - \sigma_i\boldsymbol{I} = \boldsymbol{P}\boldsymbol{D}^-\boldsymbol{P}^t = \boldsymbol{Q}\boldsymbol{D}^+\boldsymbol{Q}^t$ where $\boldsymbol{P}$ is lower diagonal with 1 s on the diagonal, and $\boldsymbol{Q}$ is upper diagonal with 1s on the diagonal, and $\boldsymbol{D}^-, \boldsymbol{D}^+$ are diagonal.
2. Calculate vector $\boldsymbol{\gamma}_k = \boldsymbol{D}^-_{kk} + \boldsymbol{D}^+_{kk} - (a_k^2 + b_{k-1}^2 - \sigma_i^2)$
3. Find minimum over k of gamma, call it k^*
4. solve the twisted factorization problem $\boldsymbol{N}^t_{k^*}\tilde{\boldsymbol{x}}_i = \boldsymbol{e}_{k^8}$, where N_k is given below.
5. Perform one more backsolve for accuracy of the system $N_{k^*}D_{k^*}N^t_{k^*}x_i = \tilde{x_i}$.

6. Find right singular vector by doing $y_i = \frac{Bx_i}{\sigma_i}$
7. Get the right and left singular vectors of A from the ones for B by applying the householder reflections from the bidiagonalization procedure.

In essence, 2 inverse power iterations are performed in a specific way with a specific initial guess. The first iteration is $(B^tB - \sigma_i^2 I)x^1 = N_k D_k N_k^t x^1 = \gamma_k x_k$ where e_k is the standard basis vector, and γ_k is chosen so that $x_k^1 = 1$, and

$$N_k = \begin{pmatrix} 1 & & & & & & \\ p_1 & \ddots & & & & & \\ & \ddots & 1 & & & & \\ & & p_{k-1} & 1 & q_k & & \\ & & & & 1 & \ddots & \\ & & & & & \ddots & q_{m-1} \\ & & & & & & 1 \end{pmatrix}$$

$$D_k = \text{diag}(D1_1, \ldots, D1_{k-1}, \gamma_k, D2_{k+1}, \ldots, D2_m)$$

The value of γ_k can be determined from the elements of the forward and backward factorization as $\boldsymbol{\gamma}_k = \boldsymbol{D}_{kk}^- + \boldsymbol{D}_{kk}^+ - (a_k^2 + b_{k-1}^2 - \sigma_i^2)$. By then choosing to twist around the index k^* that minimizes γ_k, we ensure [6] that our initial guess is closely aligned with the singular vector being calculated, increasing accuracy. An additional advantage of the twisted factorization is that the system

$$N_k D_k N_k^t x^1 = \gamma_k e_k$$

is equivalent to

$$N_k^t x^1 = e_k$$

which saves us one linear backsolve on our first iteration.
Aftersolving for x^1, we do one additional solve

$$N_k D_k N_k^t x^2 = x^1$$

And then normalize

$$x = \frac{x^2}{||x^2||}$$

and we have calculated the vector x which is a right singular vector of the diagonal matrix B corresponding to the singular value σ_i. To get the left singular vector, we simply calculate $y = \frac{Bx}{\sigma_i}$ which takes one bidiagonal multiplication.
Now it remains to "undo" the householder reflections to get from x and y to singular vectors u and

v of the dense matrix A. We do this by applying the householder reflectors in opposite order to our bidiagonal singular vector.

$$v = (I - 2v_1v_1^t)\dots(I - 2v_nv_n^t)x$$

$$u = (I - 2u_1u_1^t)\dots(I - 2u_nu_n^t)y$$

Which completes the algorithm.

4.2 Performance

In this section, we define p to be the minimum of m and n. All of the results are obtained by testing on a computer with an AMD FX-8150 Processor, 16 Gigabytes of Ram, and a Radeon HD 7770 GPU with 1G of memory.

The calculation of singular vectors of the bidiagonal matrix is not well suited to be done on the GPU, because for each singular vector we require $O(p)$ calculations, each with dependencies. In addition, we must have room to store the elements of the forward and backward factorizations, as well as γ, which means we would spend a lot of time moving data to and from global memory.

Even the calculation of γ, which has a factor of p^2 parallelism, is not well suited for the GPU. This is because it requires the diagonals of the forward and backward factorizations for each singular value. This requires us to move $2p^2$ doubles to the device, and then p^2 doubles back from the device, which is expensive for the relatively small amount of computation we end up doing there (about 7 flops per work item).

Because of the above factors, the twisted factorization was parallelized using only OpenMP, which provided some speedup.

Table 2: Twisted Factorization Performance in Gigaflops

Size	$1k \times 1k$	$2k \times 2k$	$4k \times 4k$	$8k \times 8k$
Serial	.166843	.175175	.187305	
OpenMP, 8 Threads	.358352	.44066	.4911	.5317

In Table 2, it is evident that the parallelization could be improved, as it does not achieve close to the multiple of 8 increase one could hope to see. This in in part due to the chip architechture, where each 2 cores share an FPU and part of the pipeline, causing some competition among threads. It is also in part due to the multiple parallel sections in the code which each cost some overhead. The algorithm could in principle all be done in one parallel region, giving increased speed. However, the twisted factorization is an $O(p^2)$ algorithm in total, and applying the householder reflections to get the singular vectors of A takes $O(p * (m^2 + n^2))$ operations. As expected, for a 1000×1000

matrix, $\approx 95\%$ of the singular vector calculation time is spent applying the householder reflections. This percentage only gets larger with matrix size. Because of this, most of the speedup effort will be focused on the householder reflection section.

First the Householder reflections are applied to different singular vectors seperately in parallel using OpenMP. This provided substantial speedup.
The Householder reflections applied to Y require accessing the reflectors from A that are lined up in non-contiguous memory. By transposing A into a new variable, and using that to apply the householder reflections on the right singular vectors, we get significant speedup. The transpose itself is an order mn algorithm, and indeed it takes less than a percent of the total singular value time when working on a 2000×2000 matrix. Loop unrolling was tried in the dot product and application of the householder reflections, but it had an insignificant effect.

Table 3: Householder Application Performance in Gigaflops

Size	$1k \times 1k$	$2k \times 2k$	$4k \times 4k$	$8k \times 8k$
Serial	.46435	.180354	.15176	
OpenMP, 8 threads	1.6105	1.0148	.9658	
OpenMP + Transpose, 8 threads	3.4345	2.8223	2.8486	3.1250
OpenMP + Transpose, 4 threads	2.2545	1.8157	1.8444	2.0512
OpenMP + Transpose, 2 threads	1.3404	1.2305	1.6853	

5 Conclusion

We have presented a ground-up, parallel implementation of an SVD algorithm. We have seen strong improvements over our original serial algorithm but we fall short in comparison to larger scale implementations. There are many areas of our code which could benefit from further tuning and design changes and we have presented some possibilities above. The current code is documented and available freely under the MIT license at `https://github.com/Skwaap/ddc-svd` .

6 Acknowledgments

The authors would like to thank Andreas Kloeckner for many helpful discussions.

References

[1] Lloyd N. Trefethen and David Bau III, *Numerical linear algebra*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997. MR1444820 (98k:65002)

[2] Fangbin Liu and Frank J. Seinstra, *GPU-based parallel householder bidiagonalization*, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 288–291, DOI 10.1145/1851476.1851512, (to appear in print).

[3] G. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, SIAM J. Numer. Anal., 1965, pp. 205–224.

[4] Taro Konda and Yoshimasa Nakamura, *A new algorithm for singular value decomposition and its parallelization*, Parallel Computing **35** (2009), no. 6, 331 - 344, DOI 10.1016/j.parco.2009.02.001.

[5] A. Melman, *Analysis of third-order methods for secular equations*, Math. Comp. **67** (1998), no. 221, 271–286, DOI 10.1090/S0025-5718-98-00884-9. MR1432130 (98c:65061)

[6] Christof Vömel and Jason Slemons, *Twisted factorization of a banded matrix*, BIT **49** (2009), no. 2, 433–447, DOI 10.1007/s10543-009-0217-0. MR2507610 (2010c:65042)