Large Sparse SVD in CUDA

1 Abstract

In this project report, we review the theory of the Singular Value Decomposition (SVD), applications of the SVD to Principal Component Analysis (PCA), different algorithms for computing the SVD, and state-of-the-art computational implementations for the SVD. We also present a CUDA implementation using cuBLAS and cuSPARSE for Golub-Kahan-Lanczos Bidiagonalization procedure, which is the most expensive step in computing the SVD of large sparse matrices. Our results show that our CUDA implementation has significant speedup over the state-of-the-art serial SVD software on the large-scale real-world MovieLens database.

2 The Singular Value Decomposition

The Singular Value Decomposition (SVD) of a rectangular matrix $A \in \mathbb{R}^{m \times n}$ is given by $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m \times n}$ is orthogonal, $\Sigma \in \mathbb{R}^{n \times n}$ is diagonal, and $V \in \mathbb{R}^{n \times n}$ is orthogonal [4]. The columns of U, also known as the left singular vectors, form an orthogonal basis for the column space of A, while the columns of V, also known as the right singular vectors, form an orthogonal basis for the column space of A^T. The SVD of a matrix A is related to the eigenvalue decomposition of $A^T A$ because the eigenvectors of $A^T A$ are the columns of V and the eigenvalues of $A^T A$ are the square of the singular values of A. Another useful property of the SVD is due to the Eckhart-Young Theorem, which says that if $k < r$, where r is the rank of A, and $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$ where the singular values σ_i are of decreasing magnitude, the u_i are the columns of U, and the v_i are the columns of V, then A_k is the closest rank-k approximation to A in the 2-norm and the Frobenius norm. If the singular values of the matrix decay rapidly, then approximating A by A_k for a small k will significantly reduce storage requirements while maintaining a very good approximation of the original matrix.

3 Principal Component Analysis

Principal Component Analysis (PCA) was invented in 1901 by Karl Pearson in the context of statistical correlation theory. Let a dataset be represented by a matrix, where a data point is represented by column in that matrix. For instance, the Netflix matrix of users and ratings has each column represent a user's set of ratings for each movie. The first principal component of a dataset accounts for as much variability in the data as possible, and each next principle component has the highest possible variance in a direction orthogonal to all preceding components. It turns out that PCA can be computed using the SVD. Typically, only the first $k << n$ principal components are needed because

it is know that a small number of orthogonal vectors can describe the variance in the data very well. For the Netflix matrix, it has been shown that peoples ratings can often be described by a small low-rank subset of factors, like preference for historical drama, psychological thrillers, or comedy movies starring Vince Vaughn. Computing the top k principal components corresponds to computing the top k singular triplets: σ_i, u_i, v_i for $i = 1, \ldots, k$. As we will see, there are different ways of computing the SVD: some compute all n singular triplets, while some allow you to compute just a subset of $k << n$. For PCA, computing the full SVD is would be followed by discarding the non-principal components. Thus, it would be more efficient to only compute the principal components necessary in the first place. Related problems such as Noisy Matrix Completion and Noisy Robust PCA require computing the top k singular triplets at each step of an iterative method for convex constrained optimization such as Accelerated Proximal Gradient [8].

4 Algorithms for Computing the SVD

4.1 Dense SVD Frameworks

Many SVD algorithms start by Householder reduction to bidiagonal form. Then using the Golub-Kahan step, Demmel-Kahan step, Bisection step, or the Divide and Conquer method of Gu and Eisenstat (1995), it is possible to compute the singular values and singular vectors of the bidiagonal matrix [3]. Combining the singular vectors with the Householder reduction, we can form the SVD of the matrix as a whole. Other methods such as Biorthogonalization and Jacobi Rotation iterate directly on the input matrix [3]. All of these methods require the whole matrix to be in memory and can only compute the full SVD.

4.2 Sparse SVD Frameworks

Instead of assuming that the whole matrix is in memory, let's assume that we have an efficient way of applying our matrix A to a vector x. This can be a sparse matrix vector multiplication, or any other way of applying A to a vector which takes less than $O(mn)$ flops. The Lanczos procedure builds an orthogonal basis for the k dimensional Krylov subspace of A given by $\text{span}\{r, Ar, \ldots, A^{k-1}r\}$ for some vector r with just matrix vector products. The Golub-Kahan-Lanczos Bidiagonalization procedure uses this approach to reduce any rectangular matrix A to bidiagonal form without the hassle of requiring dense intermediate matrices, as is required by the Householder reduction to bidiagonal form. Even further, Golub-Kahan-Lanczos Bidiagonalization can compute top k singular triplets for any $k \leq n$, so it doesn't need to compute the full SVD. This bidiagonalization procedure would then be followed by an algorithm for computing the singular values and singular vectors of a bidiagonal matrix, such as the Divide and Conquer method of Gu and Eisenstat, which is considered state-of-the-art.

Algorithm 1 describes the Golub-Kahan-Lanczos Bidiagonalization procedure. The input consists of a matrix $A \in \mathbb{R}^{m \times n}$ where $A(1{:}n, 1{:}n)$ is nonsingular, a unit vector $u_c \in \mathbb{R}^n$ and the number of desired singular triplets $K \leq n$. Output is $\alpha \in \mathbb{R}^k, \beta \in \mathbb{R}^k, U \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{k \times k}, p \in \mathbb{R}^m$, such that $AV(:, 1{:}k) = U(:, 1{:}k)B(1{:}k, 1{:}k) + p_k e_k^T$ where $1 \leq k \leq K, B = \text{spdiags}([\alpha, \beta], [0, -1], k, k)$ and e_k is the k-th column of the k by k identity matrix. The first column of U is u_c. Notice that most of the work goes into matrix-vector multiplication at lines 5 and 8.

Algorithm 1 Golub-Kahan-Lanczos Bidiagonalization

procedure GKL-BIDIAG(A, u_c, K)
  $k := 1, p_0 = u_c, \beta_1 = 1, v_0 = 0$
  while $\beta_k > 0, k < K$ **do**
    $u_k := p_{k-1}/\beta_k$
    $r_k := A^T u_k - \beta_k v_{k-1}$ ▷ Matrix-vector multiply
    $\alpha_k := \|r_k\|_2$
    $v_k := r_k/\alpha_k$
    $p_k := Av_k - \alpha_k u_k$ ▷ Matrix-vector multiply
    $\beta_{k+1} := \|p_k\|_2$
    $k := k + 1$
  end while
  return $\alpha, \beta, U = [u_1|\cdots|u_k], V = [v_1|\cdots|v_k], p_k$
end procedure

5 Current SVD Implementations

Both of the dense categories described below require the entire matrix in memory and can only compute the full SVD. The sparse category only requires a function for matrix vector multiplication and the software mentioned is best suited for computing the largest $k << n$ singular triplets of large sparse matrices.

5.1 Dense Serial SVD Software

State-of-the-art routines for the SVD of a dense matrix include LAPACK [2] and Intel MKL [10] implementations of xGESVD (Householder reduction to bidiagonal form followed by Demmel-Kahan bidiagonal SVD) and xGESDD (Householder reduction to bidiagonal form followed by Divide and Conquer bidigaonal SVD). xGESVD uses less memory than xGESDD, but xGESDD is typically faster. MATLAB decides internally whether to use xGESVD or xGESDD in the svd function. LAPACK is free research software while Intel MKL is proprietary.

5.2 Dense Shared Memory Parallel SVD Software

In shared memory, MAGMA [1] and CULA [5] implement xGESVD. They both use cuBLAS [9] to speedup the corresponding dense serial algorithms. MAGMA is completely free and open source research software while CULA is commercial and closed source. Currently, these packages just call xGESVD when xGESDD is called, so implementing xGESDD in shared memory appears to be an open research problem.

5.3 Sparse SVD Software

Both ARPACK [7] and PROPACK [6] can use an operator for matrix vector multiplication instead of storing the whole matrix in memory, and are both considered state-of-the-art software for computing the largest $k << n$ singular triplets of large sparse matrices. ARPACK, which called by MATLAB's eigs and svds functions, is designed to solve eigenvalue problems, but it can be adapted for SVD computation with some post-processing. ARPACK has a distributed memory Fortran implementation. PROPACK

is specifically for SVD computations and consistently outperforms ARPACK for large sparse SVD problems in serial.

6 Sparse SVD Implementation in CUDA

Given the three categories in the previous section, one would expect that there should also be a section for Sparse Shared Memory Parallel SVD computation. However, after doing some research, I could not find a standard state-of-the-art software package for large sparse SVD computations in shared memory. Hence, I present a CUDA implementation the Golub-Kahan-Lanczos Bidiagonalization procedure to allow for computation of the top k singular triplets of large sparse matrices. I make significant use of the cuBLAS and cuSPARSE libraries from NVIDIA. I use cublasSscal for scaling a vector by a scalar, cublasScopy to copy a vector into another vector, cusparseScsrmv for sparse matrix vector multiplication using Compressed Sparse Row storage of the matrix, cublasSaxpy for adding a vector by a scaled vector, and cublasSnrm2 to get the 2-norm of a vector. The core of the implementation, excluding malloc's and memcpy's, is below:

```
beta[0] = 1;
int k = 0;
// Begin iteration
while (k < K && beta[k] > 0)
{
  if (k > 0)
  {
    // U(:,k) = p/beta(k);
    work = 1/beta[k];
    cublasStatus = cublasDscal(cublasHandle, N, &work, d_p, 1);
    cublasStatus = cublasDcopy(cublasHandle, M, d_p, 1, &d_U[k*M], 1);
    // r = A'*U(:,k) - beta(k)*V(:,k-1);
    cusparseDcsrmv(cusparseHandle, CUSPARSE_OPERATION_TRANSPOSE, M, N, nz,
                   &one, descr, d_val, d_row, d_col, &d_U[k*M], &zero, d_r);
    work = -beta[k];
    cublasStatus = cublasDaxpy(cublasHandle, N, &work, &d_V[(k-1)*N], 1, d_r, 1);
  } else {
    // U(:,k) = p;
    cublasStatus = cublasDcopy(cublasHandle, M, d_p, 1, &d_U[k*M], 1);
    // r = A'*U(:,k);
    cusparseDcsrmv(cusparseHandle, CUSPARSE_OPERATION_TRANSPOSE, M, N, nz,
    &one, descr, d_val, d_row, d_col, &d_U[k*M], &zero, d_r);
  }
  // alpha(k) = norm(r);
  cublasDnrm2(cublasHandle, N, d_r, 1, &alpha[k]);
  // V(:,k) = r/alpha(k);
  work = 1/alpha[k];
  cublasStatus = cublasDscal(cublasHandle, N, &work, d_r, 1);
  cublasStatus = cublasDcopy(cublasHandle, N, d_r, 1, &d_V[k*M], 1);
  // p = A*V(:,k) - alpha(k)*U(:,k);
  cusparseDcsrmv(cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, M, N, nz,
```

```
                &one, descr, d_val, d_row, d_col, &d_V[k*M], &zero, d_p);
  work = -alpha[k];
  cublasStatus = cublasDaxpy(cublasHandle, N, &work, &d_U[k*N], 1, d_p, 1);
  // if k < n
  if (k < N)
  {
    // beta(k+1) = norm(p);
    cublasDnrm2(cublasHandle, N, d_p, 1, &beta[k+1]);
  }
  // end
  cudaThreadSynchronize();
  printf("iteration = %d\n", k);
  // k = k + 1;
  k++;
}
```

Table 1: Random Examples (top $k = 100$ triplets)

m	n	nnz	$s = \text{nnz}/mn$	ARPACK	PROPACK	GKLB
128	128	382	0.023	0.0777 (s)	0.0711 (s)	2.31e-8 (s)
1,024	1,024	3070	0.0029	0.355 (s)	0.352 (s)	2.58e-8 (s)
1,048,576	1,048,576	3,145,726	2.86e-6	1,659.68 (s)	OOM	2.3e-6 (s)

Table 2: MovieLens Datasets (top $k = 300$ triplets)

Matrix	m	n	nnz	$s = \text{nnz}/mn$	ARPACK	PROPACK	GKLB
ml-100K	943	1,682	100,000	0.063	2.58 (s)	1.36 (s)	7.20e-7 (s)
ml-1M	6,040	3,900	1,000,209	0.045	31.4 (s)	13.4 (s)	1.34e-5 (s)
ml-10M	71,567	10,681	10,000,054	0.013	280.3 (s)	84.2 (s)	2.38e-5 (s)

7 Benchmarks and Results

For the large sparse real-world datasets that I focus on in this report, dense algorithms for the SVD will run out of memory very quickly. Thus I will only be comparing the algorithm presented in this report with the state of the art algorithms for large sparse SVD: ARPACK (which is called by svds in MATLAB) and PROPACK (an independent, easy to use MATLAB package).

Table 1 shows a summary of results on random tridiagonal matrices. I generated random tridiagonal matrices of the same dimension in both my CUDA implementation and in MATLAB svds.

Table 2 shows a summary of the real-world datasets I will be running on. For the real-world data, I used the files from NIST called mmio.c and mmio.h to read a file in Matrix Market format. I used cusparseXcoo2csr to convert the matrix from Coordinate format (arrays of equal length for rows, cols, and values), to Compressed Sparse Row format

My main result so far is that to get the top 300 principal components of the MovieLens datasets took on the order of seconds with PROPACK, while the bidiagonalization procedure in CUDA was on the order of microseconds on the Jinx cluster, including time to copy memory to device and back. In MATLAB (calling LAPACK) I found the SVD of a 100 by 100 bidiagonal matrix takes 0.0636 seconds (which would be done after GKLB for the Random Examples above), while SVD of a 300 by 300 bidiagonal matrix takes 0.1396 seconds (which would be done after GKLB for the MovieLensDataset Examples above). This step is now the bottleneck to compute the SVD for large sparse matrices, and may be the subject of future work.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. Siam, 1999.

[3] Alan Kaylor Cline and Inderjit S Dhillon. Computation of the singular value decomposition. *Handbook of linear algebra*, pages 45–1, 2006.

[4] Gene H Golub and Charles F Van Loan. *Matrix Computations*, volume 4. JHU Press, 2012.

[5] John R Humphrey, Daniel K Price, Kyle E Spagnoli, Aaron L Paolini, and Eric J Kelmelis. Cula: hybrid gpu accelerated linear algebra routines. In *SPIE Defense, Security, and Sensing*, pages 770502–770502. International Society for Optics and Photonics, 2010.

[6] Rasmus Munk Larsen. Propack-software for large and sparse svd calculations. *Available online. URL http://sun.stanford.edu/rmunk/PROPACK*, 2004.

[7] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, volume 6. Siam, 1998.

[8] Lester W Mackey, Michael I Jordan, and Ameet Talwalkar. Divide-and-conquer matrix factorization. In *Advances in Neural Information Processing Systems*, pages 1134–1142, 2011.

[9] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.

[10] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.