Large Sparse SVD in CUDA

Joseph Vokt, CSE 6230, Georgia Tech (joseph.vokt@gatech.edu)

Abstract

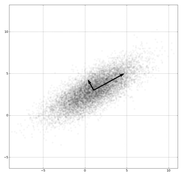
For this project, we review the theory of the Singular Value Decomposition (SVD), applications of the SVD to Principal Component Analysis (PCA), different algorithms for computing the SVD, and state-of-the-art computational implementations for the SVD. We also present a CUDA implementation using cuBLAS 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.

The Singular Value Decomposition

- SVD of $A \in \mathbb{R}^{m \times n}$: $A = U\Sigma V^T$, where $U^TU = I, V^TV = I$ and Σ is diagonal
- Related to Eigenvalue Decomposition
- Useful for finding a low-rank approximation of optimal proximity

Principal Component Analysis

- PCA is a truncated SVD of a data matrix (e.g. the Netflix Matrix)
- Columns of the data matrix are points in high dimensional space which PCA reduces to lower dimensional space:



Algorithms for Computing the SVD

- Dense SVD Frameworks: reduce to bidiagonal form with Householder transforms, then find the SVD of the bidiagonal matrix: $A = U_k B_k V_k^T = U_k(U_B \Sigma V_B^T)V_k^T = U\Sigma V^T$, where $B_k \in \mathbb{R}^{k \times k}$ is bidiagonal.
- Sparse SVD Frameworks: reduce to bidiagonal form with Golub-Kahan-Lanczos Bidiagonalization, then find the SVD of the bidiagonal matrix.

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**
    $u_k := p_{k-1}/\beta_k$
    $r_k := A^T u_k - \beta_k v_{k-1}$ ▷ Sparse matrix-vector multiply
    $\alpha_k := \|r_k\|_2$
    $v_k := r_k/\alpha_k$
    $p_k := Av_k - \alpha_k u_k$ ▷ Sparse 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

Current SVD Implementations

- Dense Serial: LAPACK, Intel MKL have xGESVD, xGESDD. Called by MATLAB svd.
- Dense Shared Memory Parallel: MAGMA, CULA have xGESVD
- Sparse Serial: ARPACK is called by MATLAB svds, PROPACK implements lansvd

Sparse SVD Implementation in CUDA

- cuBLAS
  - cublasDscal(cublasHandle_t handle, int n, const double *alpha, double *x, int incx):
    Scales the vector x by the scalar alpha and overwrites it with the result.
  - cublasDcopy(cublasHandle_t handle, int n, const double *x, int incx, double *y, int incy):
    Copies the vector x into the vector y.
  - cublasDaxpy(cublasHandle_t handle, int n, const double *alpha, const double *x, int incx, double *y, int incy):
    Multiplies the vector x by the scalar alpha and adds it to the vector y.
  - cublasDnrm2(cublasHandle_t handle, int n, const double *x, int incx, double *result):
    Computes the Euclidean norm of the vector x.
- cuSPARSE
  - cusparseXcoo2csr(cusparseHandle_t handle, const int *cooRowInd, int nnz, int m, int *csrRowPtr, cusparseIndexBase_t idxBase):
    Converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).
  - cusparseDcsrmv(cusparseHandle_t handle, cusparseOperation_t transA, int m, int n, int nnz, const double *alpha, const cusparseMatDescr_t descrA, const double *csrValA, const int *csrRowPtrA, const int *csrColIndA, const double *x, const double *beta, double *y):
    Performs y = alpha*op(A)*x+beta*y where A is an m by n sparse matrix that is defined in CSR storage format by the three arrays csrValA, csrRowPtrA, and csrColIndA.

Benchmarks and Results

- 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)

- 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)

Note: SVD of a 100 by 100 bidiagonal matrix in MATLAB (using LAPACK) takes 0.0636 seconds, while SVD of a 300 by 300 bidiagonal matrix takes 0.1396 seconds. This step is now the bottleneck to compute the SVD for large sparse matrices, and may be the subject of future work.