Randomized low rank SVD algorithm implementation for single core, multi core, and GPU architectures

SV and PGM

December 27, 2014

Abstract

This document describes the implementation in C of a randomized low rank SVD algorithm. The implementation consists of three different codes: for single core, multi core, and GPU architectures. For each case, we describe some details of the implementation and then go on to compare their performance for matrices of different size.

1 Introduction

Any matrix $A \in \mathbb{R}^{m\times n}$ has the SVD decomposition such that:

$$A = \sum_{i=1}^{\min(m,n)} \sigma_i u_i v_i^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T = U\Sigma V^T \quad ; \quad \Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_r, 0, \ldots, 0) \in \mathbb{R}^{m\times n},$$

where U and V are orthogonal matrices ($U^TU = UU^T = V^TV = VV^T = I$) and where $\sigma_j = 0$ for $j > r$ where $r = \mathrm{rank}(A) \leq \min(m, n)$. The sizes of the matrices are: U is $m \times m$, Σ is $m \times n$ and V is $n \times n$. There are r nonzero singular values: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$. In many situations, the rank of the matrix A may be smaller than $\min(m, n)$ so that $\sigma_j = 0$ for $j = (r+1), \ldots, \min(m, n)$ and certainly many σ_j with $j < r$ may still be very small. Pictorially, the decomposition of A takes the form:

$$A = \begin{bmatrix} u_1 \ldots u_r & u_{r+1} \ldots u_m \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & \sigma_2 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & \ldots & \sigma_r & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \ldots & 0 & 0 \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_r^T \\ v_{r+1}^T \\ \vdots \\ v_n^T \end{bmatrix} = \begin{bmatrix} u_1 \ldots u_r \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \ldots & 0 \\ 0 & \sigma_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \sigma_r \end{bmatrix} \begin{bmatrix} v_1^T \\ \vdots \\ v_r^T \end{bmatrix}$$

For a matrix which is not well conditioned, many nonzero singular values σ_j will be very small relative to the largest singular value σ_1 and the drop off in value starting from σ_1 will be rapid and nonlinear.

We define the low rank SVD of a matrix by taking into account only the first k singular values and vectors:

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T = U_k \Sigma_k V_k^T \approx U \Sigma V^T$$

with $k < r$, U_k being $m \times k$, Σ_k being $k \times k$, and V_k being $n \times k$. When $k << r \leq \min(m, n)$ these matrices are significantly smaller than the corresponding full SVD matrices U, Σ, and V. When the matrix A is of low numeric rank, this approximation can be accurate even for small k. In any case, the matrix A_k is the best rank k approximation of A with the bound:

$$||Az - A_k z||_2 \leq \sigma_{k+1} ||z||_2 \quad \forall z \in \mathbb{R}^n$$

The choice of rank k is up to the user, but greater k requires greater computation time and memory requirements for the components U_k, Σ_k, and V_k.

2 Randomized Algorithm

We can compute the low rank SVD components U_k, Σ_k, and V_k by taking the first k columns of the corresponding quantities of the full SVD, but that is prohibitively expensive for large matrices. Instead, we make use of a much more efficient approximate randomized method proposed in [1]. The randomized algorithm consists of the following steps:

- Take k samples of the range of A by multiplying A by random Gaussian vectors to form sample matrix Y of size $m \times k$ such that $\operatorname{range} Y \approx \operatorname{range} A$.
- Orthogonalize this set of samples forming the matrix Q.
- Project the original matrix into a lower dimensional one: $B = Q^T A$ where B is $k \times n$, substantially smaller than A, which is $m \times n$.
- Take the SVD of the smaller matrix $B = \tilde{U}_k \Sigma_k V_k^T$.
- Take as low rank SVD of A the product $U_k \Sigma_k V_k^T$ with $U_k = Q\tilde{U}_k$ (since $QQ^T A \approx A$).

Since the matrix B of size $k \times n$, can still be quite large for large n, we may wish to implement the above steps in a different way to avoid taking the SVD of a large matrix. We proceed to describe three different variants of the randomized algorithm below, which are implemented in our software package.

2.1 Version I

The first modification of the original algorithm computes U_k and V_k without having to take the SVD of the $k \times n$ matrix B. Instead, the eigendecomposition of the the smaller $k \times k$ matrix BB^T is used.

We use the relations:

$$B = \tilde{U}_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T \quad ; \quad B^T = V_k \Sigma_k \tilde{U}_k^T \quad ; \quad B v_i = \sigma_i \tilde{u}_i$$

$$BB^T = \left(\sum_{i=1}^{k} \sigma_i \tilde{u}_i v_i^T \right) \left(\sum_{j=1}^{k} \sigma_j \tilde{u}_j v_j^T \right)^T = \sum_{i,j=1}^{k} \sigma_i \sigma_j \tilde{u}_i v_i^T v_j \tilde{u}_j^T = \sum_{i=1}^{k} \sigma_i^2 \tilde{u}_i \tilde{u}_i^T = \tilde{U}_k D \tilde{U}_k^T$$

To compute the right eigenvectors v_i, we can use the following relations:

$$B^T \tilde{U}_k = V_k \Sigma_k \tilde{U}_k^T \tilde{U}_k = V_k \Sigma_k \implies V_k = B^T \tilde{U}_k \Sigma^{-1}$$

assuming all the singular values in Σ_k are above zero (which is the case for k smaller than the numerical rank). The Matlab like pseudocode for this implementation proceeds as follows:

```
function [U,Sigma,V] = rsvd_version1(A,k)
    m = size(A,1);
    n = size(A,2);

    R = randn(n,k);
    Y = A*R;

    Q = orth(Y);

    B = Q'*A;
    Bt = B';

    BBt = B*Bt;

    [Uhat,D] = eig(BBt);

    Sigma = sqrt(D);
    U = Q*Uhat;

    V = Bt*Uhat*Sigma^{-1};
end
```

Note that this implementation is quite simple. The most complicated operation is the eigendecomposition of the symmetric matrix BB^T. Since this matrix is small ($k \times k$), the eigendecomposition is not expensive. The functionality to extract the eigenvalues and eigenvectors of a symmetric matrix can be found in many numerical linear algebra packages. Otherwise, one can use a Lanczos type routine to reduce the matrix to tridiagonal form and then perform repeated QR factorizations to extract the eigenvectors. However, one must be careful about orthogonality and efficiency of such implementations. The disadvantage of this version is that working with the matrix BB^T essentially squares the condition number of A, such that small singular values near machine precision may not be properly resolved. This is an issue only if A is expected to have very small singular values amongst $\sigma_1, \ldots, \sigma_k$.

2.2 Version II

Here, we use a QR factorization of B^T instead of forming BB^T. Out of this we get a $k \times k$ matrix R on which we perform the SVD. This version is based on the following calculations. Let us take the (economic form of the) QR factorization $B^T = \hat{Q}\hat{R}$. Then $\hat{R}$ is $k \times k$ and taking the SVD yields $\hat{R} = \hat{U}\hat{\Sigma}\hat{V}^T$.

$$A \approx QQ^TA = QB = Q\hat{R}^T\hat{Q}^T = Q\hat{V}\hat{\Sigma}\hat{U}^T\hat{Q}^T$$

Thus, the low rank SVD components are $U_k = Q\hat{V}, \Sigma_k = \hat{\Sigma}, V_k = \hat{Q}\hat{U}$. The Matlab like pseudocode for this implementation proceeds as follows:

```
function [U,Sigma,V] = rsvd_version2(A,k)
    m = size(A,1);
    n = size(A,2);

    R = randn(n,k);
    Y = A*R;

    Q = orth(Y);

    Bt = A'*Q;

    [Qhat,Rhat] = qr(Bt,'0');

    [Uhat,Sigma,Vhat] = svd(Rhat);

    U = Q*Vhat;
    V = Qhat*Uhat;
end
```

2.3 Version III

This version of the algorithm is a slight modification of Version II and is effective in cases when the singular values of the matrix decrease slowly. This does not necessarily imply that the matrix is well conditioned. Rather, imagine a matrix A with singular values $\sigma_1, \ldots, \sigma_n$ where the ratio between the first and last singular value is large. However, there maybe a sequence of singular values $\sigma_{i_1}, \ldots, \sigma_{i_p}$ such that $1 \leq i_1 \leq i_p \leq n$ for which the singular values fall of slowly in value. In such case, it is better to use a different method in order to approximate the range of the matrix A. In Versions I and II, the matrix Q which we use to project the matrix A into a lower dimensional space is constructed as follows:

$$Y = AR \rightarrow Q = orth(Y)$$

If we don't perform any orthogonalization, then $Q = Y$ and we sample the range of the original matrix A. Instead, based on observations in [1], we would like to work with the matrix $\left((AA^T)^qA\right)R$, where

q is some integer greater than one. Note first that A and $(AA^T)^q A$ have the same eigenvectors and related eigenvalues. Plugging in the SVD $A = U\Sigma V^T$ we have:

$$AA^T = U\Sigma^2 U^T \implies (AA^T)^2 = U\Sigma^2 U^T U\Sigma^2 U^T = U\Sigma^4 U^T \implies (AA^T)^q = U\Sigma^{2q} U^T$$
$$\implies (AA^T)^q A = U\Sigma^{2q} U^T U\Sigma V^T = U\Sigma^{2q+1} V^T$$

The matrix $Z = (AA^T)^q AR$ can be built up by means of the following iterative procedure:

```
Z_0 = A R;
for j=1:q
    Y_j = A^T Z_{j-1};
    Z_j = A Y_j = A A^T Z_{j-1}
end
```

This implies $Z_j = (AA^T)^j Z_0 = (AA^T)^j AR$. Note that in practice, we need to orthonormalize between the computations to prevent computing M^j with M a matrix having singular values greater than one. Thus, the construction of Q used to project the matrix A into a lower dimensional space proceeds as follows, where we orthogonalize every two iterations:

```
Z_0 = A R; Q_0 = orth(Z_0);
for j=1:q
    Y_j = A^T Q_{j-1};
    W_j = orth(Y_j); % done for even j
    Z_j = A W_j;
    Q_j = orth(Z_j); % done for even j
end
Q = orth(Z_q); % if q not even
```

3 Developed Software

In this section, we describe the developed software which has been written to implement Versions I, II, and III of the randomized SVD algorithm. We have developed three different codes: for single core, multi-core, and GPU archtectures. In each case, we have used well known software libraries to implement BLAS (matrix-matrix and matrix-vector operations) and QR, eigendecomposition, and SVD operations. Each of the codes is written using the C programming language. Each code can load a matrix from disk stored using the following simple binary format:

```
num_rows (int)
num_columns (int)
nnz (double)
...
nnz (double)
```

where the nonzeros are listed in order of double loop over the rows and columns of the matrix. Note that even zero values are written in this format. This format is thus best for dense matrices. The

matrix can be loaded using the supplied matrix_load_from_binary_file(char *fname) function. For each code, the low rank SVD can be computed by Version I, II, or III of the algorithm via the following basic call sequence (illustrated here for the single core code):

```
// load matrix
gsl_matrix *M = matrix_load_from_binary_file(mfile);
m = M->size1; n = M->size2;

// set svd rank (< min(m,n))
k = 300;

// set up SVD components
gsl_matrix *U = gsl_matrix_calloc(m,k);
gsl_matrix *S = gsl_matrix_calloc(k,k);
gsl_matrix *V = gsl_matrix_calloc(n,k);

// call random SVD (use one of three variants)
randomized_low_rank_svd1(M, k, U, S, V);
randomized_low_rank_svd2(M, k, U, S, V);
randomized_low_rank_svd3(M, k, q, U, S, V);
```

3.1 Single Core Code

The single core code is written using functions from the GNU Scientific Library (GSL). It is a commonly used well documented library that has well optimized functions. The code uses gsl_vector and gsl_matrix objects from the library. Matrix-matrix and matrix-vector operations are done using the included BLAS functionality of GSL. For example, a call to a matrix-vector multiply looks like:

```
/* y = M*x */
void matrix_vector_mult(gsl_matrix *M, gsl_vector *x, gsl_vector *y){
    gsl_blas_dgemv (CblasNoTrans, 1.0, M, x, 0.0, y);
}
```

which is a simplified version of a regular BLAS call. Matrices and vectors are initilized via gsl_matix_set(...) and gsl_vector_set(...) commands. This code performs well and is somewhat faster than a similar Octave/Matlab implementation. However, it does not take advantage of multi-processor architectures and runs on a single core.

3.2 Multi Core Code

The multi-core code is written using the Intel Math Kernel Library (MKL) which includes multi-threaded functions for various matrix-vector operations. We define our own matrix and vector objects using the following simple constructions:

```
typedef struct {
    int nrows, ncols;
    double * d;
} mat;

typedef struct {
    int nrows;
    double * d;
} vec;

/* initialize new matrix and set all entries to zero */
mat * matrix_new(int nrows, int ncols)
{
    mat *M = malloc(sizeof(mat));
    M->d = (double*)mkl_calloc(nrows*ncols, sizeof(double), 64);
    M->nrows = nrows;
    M->ncols = ncols;
    return M;
}

/* initialize new vector and set all entries to zero */
vec * vector_new(int nrows)
{
    vec *v = malloc(sizeof(vec));
    v->d = (double*)mkl_calloc(nrows,sizeof(double), 64);
    v->nrows = nrows;
    return v;
}
```

Access to matrix and vector elements is then provided via the functions:

```
// column major format
void matrix_set_element(mat *M, int row_num, int col_num, double val){
    M->d[col_num*(M->nrows) + row_num] = val;
}

double matrix_get_element(mat *M, int row_num, int col_num){
    return M->d[col_num*(M->nrows) + row_num];
}


void vector_set_element(vec *v, int row_num, double val){
    v->d[row_num] = val;
}
```

```
14
15
16 double vector_get_element(vec *v, int row_num){
17     return v->d[row_num];
18 }
```

For efficiency, we avoid the use of double arrays for matrices, keeping all data in a one dimensional array. This is useful also to parallelize the code for multi-core processors. For this, we use simple OpenMP constructs. For example the frobenius norm of a matrix can be calculated using:

```
1  /* matrix frobenius norm */
2  double matrix_frobenius_norm(mat *M){
3      int i;
4      double val, normval = 0;
5      #pragma omp parallel shared(M,normval) private(i,val)
6      {
7      #pragma omp for reduction(+:normval)
8      for(i=0; i<((M->nrows)*(M->ncols)); i++){
9          val = M->d[i];
10         normval += val*val;
11     }
12     }
13     return sqrt(normval);
14 }
```

Here we have used OpenMP pragmas to split the work in the for loop across several different chunks, speeding up the calculation when several cores are available. The call to the low rank SVD is performed using a similar construction as in the GSL code. The code automatically distributes work amongst the number of available cores. Alternatively one can specify the number of threads to use via an environmental variable setting:

```
1 export OMP_NUM_THREADS=6
```

The number of threads should generally be set to the number of available cores for best performance. This code is significantly faster than the single core code for larger matrices.

3.3 GPU Code

The GPU code is written using the CULA library which imlements BLAS and LAPACK functionality using the CUDA toolkit for NVIDIA graphics cards. CULA provides two verisons of each function: a version which transfers data between host and GPU and performs calculation on the GPU, then transfers the result back to the host; and a device version, which assumes data is already on the GPU and keeps the result on the GPU after computation. The default code uses the host functions and openMP host code. That is for example, the frobenius norm of the matrix can still be performed with openMP multithreading on the host. On the other hand expensive functions like QR and SVD are evaluated on the GPU.

4 Performance Comparisons

Here we present some performance comparisons amongst the different codes. We test on two different machines, a basic two core laptop (Intel P8700 2.53 Ghz 2 core cpu) and a 6 core cluster node containing an Intel Xeon E5-2440 chip (6 cores, up to 2.90 Ghz) and an NVIDIA Tesla M2070-Q graphics card. First, we test with a 1500×2500 matrix using the laptop.

k	Single Core (s)	Multi Core (s)
100	2	1
300	16	1
600	52	2

Runtimes for V1 on Laptop

k	Single Core (s)	Multi Core (s)
100	2	1
300	17	1
600	54	2

Runtimes for V2 on Laptop

It is clear from this comparison that the MKL code, compiled with the Intel C compiler, runs much faster than the single core GSL code compiled with gcc. Thus, we go on to compare the runtime of the MKL code to the CULA code for a large dense matrix. Below, we use a $6000x12000$ matrix and different values of rank k. We run on a cluster node;

k	Multi Core (s)	GPU (s)
1000	4	6
3000	17	22
6000	69	78

Runtimes for V1 on Cluster

k	Multi Core (s)	GPU (s)
1000	5	6
3000	25	28
6000	136	116

Runtimes for V2 on Cluster

The CULA GPU code offers better performance for Version III of the algorithm, when $q >> 1$. Below, consider the runtimes for matrix A_1 of size 3000×4000 with $k = 500$ and matrix A_2 of size 7000×10000 with $k = 1000$.

q	Multi Core (s)	GPU (s)
5	3	2
10	4	3
20	7	4

Runtimes for V3 on Cluster (A_1)

q	Multi Core (s)	GPU (s)
5	17	8
10	29	12
20	54	20

Runtimes for V3 on Cluster (A_2)

References

[1] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2):217–288, January 2011.