JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 16, Number 11, 2009

Pp. 1593–1599
DOI: 10.1089/cmb.2008.0221

Parallel GPU Implementation of Iterative PCA Algorithms

M. ANDRECUT

ABSTRACT

Principal component analysis (PCA) is a key statistical technique for multivariate data analysis. For large data sets, the common approach to PCA computation is based on the standard NIPALS-PCA algorithm, which unfortunately suffers from loss of orthogonality, and therefore its applicability is usually limited to the estimation of the first few components. Here we present an algorithm based on Gram-Schmidt orthogonalization (called GS-PCA), which eliminates this shortcoming of NIPALS-PCA. Also, we discuss the GPU (Graphics Processing Unit) parallel implementation of both NIPALS-PCA and GS-PCA algorithms. The numerical results show that the GPU parallel optimized versions, based on CUBLAS (NVIDIA), are substantially faster (up to 12 times) than the CPU optimized versions based on CBLAS (GNU Scientific Library).

Key words: algorithms, automata, combinatorial optimization, statistical mechanics, stochastic processes.

1. INTRODUCTION

Principal component analysis (PCA) is one of the most valuable results from applied linear algebra, and probably the most popular method used for compacting higher dimensional data sets into lower dimensional ones for data analysis, visualization, feature extraction, or data compression (Jackson, 1991; Jolliffe, 2002). PCA provides a statistically optimal way of dimensionality reduction by projecting the data onto a lower-dimensional orthogonal subspace that captures as much of the variation of the data as possible. Unfortunately, PCA quickly becomes quite expensive to compute for high-dimensional data sets, where both the number of variables and samples is high. Therefore, there is a real need in many applications to accelerate the computation speed of PCA algorithms. For large data sets, the standard approach is to use an iterative algorithm which computes the components sequentially, and to avoid the global methods which calculate all the components simultaneously. NIPALS-PCA (Wold et al., 1987) is the most frequently used iterative algorithm, and often considered as the standard PCA algorithm. However, for large data matrices, or matrices that have a high degree of column collinearity, NIPALS-PCA suffers from loss of orthogonality, due to the errors accumulated in each iteration step (Kramer, 1998). Therefore, in practice, it is only used to estimate the first few components. Here, we address both the speed and orthogonality problems, and we offer new solutions which eliminate these shortcomings of the iterative PCA algorithms.

Institute for Biocomplexity and Informatics, University of Calgary, Calgary, Alberta, Canada.

We formulate an iterative PCA algorithm based on the Gram-Schmidt re-orthogonalization, which we called GS-PCA. This algorithm is stable from the orthogonality point of view, and if necessary, it can be used to calculate the full set of principal components. The speed-up issue is tackled with a parallel implementation for Graphics Processing Units (GPUs). Here, we present the GPU parallel implementation of both NIPALS-PCA and GS-PCA algorithms. The numerical results show that the GPU parallel optimized versions, based on CUBLAS (NVIDIA) (NVIDIA, 2008), are substantially faster (up to 12 times) than the CPU optimized versions based on CBLAS (GNU Scientific Library) (Galassi et al., 2006).

2. METHODS

2.1. Iterative principal component analysis

In the following description, the dataset to be analyzed is represented by the $M \times N$ matrix $\mathbf{X}$. Each column, $\mathbf{X}^{(n)}, n = 0, \ldots, N-1$, contains all the observations of one attribute. Also, we assume that each column is mean centered; i.e., if $\widetilde{\mathbf{X}}^{(n)}$ are the original vectors, then

$$\mathbf{X}^{(n)} = \widetilde{\mathbf{X}}^{(n)} - N^{-1} \sum_{n=0}^{N-1} \widetilde{\mathbf{X}}^{(n)} \tag{1}$$

PCA transforms the set of input column vectors $[\mathbf{X}^{(0)}| \ldots |\mathbf{X}^{(N-1)}]$ into another set of column vectors $[\mathbf{T}^{(0)}| \ldots |\mathbf{T}^{(N-1)}]$, called principal component scores. This transformation has the property that most of the original data's information content (or most of its variance) is stored in the first few component scores. This allows reduction of the data to a smaller number of dimensions, with low information loss, simply by discarding the last component scores. Each component is a linear combination of the original inputs, and each component is orthogonal. This linear transformation of the matrix $\mathbf{X}$ is specified by a $N \times N$ matrix $\mathbf{P}$ so that the matrix $\mathbf{X}$ is factorized as:

$$\mathbf{X} = \mathbf{T}\mathbf{P}^T, \tag{2}$$

where $\mathbf{P}$ is known as the loadings matrix.

There are several PCA algorithms in the literature, namely SVD (singular value decomposition) and NIPALS (nonlinear iterative partial least squares), which use the data matrix, and POWER and EVD (eigenvalue decomposition), which use the covariance of the data matrix (Jackson, 1991; Jolliffe, 2002). SVD and EVD extract all the principal components simultaneously, while NIPALS and POWER calculate them sequentially. Unfortunately, the traditional implementation of PCA through SVD or EVD quickly becomes prohibitive for very large data sets. In this case, an approximate solution can be more efficiently obtained using the iterative approach based on the NIPALS algorithm.

2.2. NIPALS-PCA algorithm

The NIPALS-PCA algorithm can be described as follows (Wold et al., 1987; Kramer, 1998). In the first step, the initial data $\mathbf{X}$ is copied into the residual matrix $\mathbf{R}$. Then, in the next steps, the algorithm extracts iteratively one component at a time ($k = 0, 1 \ldots, K \leq N$) by repeated regressions of $\mathbf{X}^T$ on scores $\mathbf{T}^{(k)}$ to obtain improved loads $\mathbf{P}^{(k)}$, and of $\mathbf{X}$ on these $\mathbf{P}^{(k)}$ to obtain improved scores $\mathbf{T}^{(k)}$. After the convergence is achieved, this process is followed by a deflation of the data matrix:

$$\mathbf{X} \leftarrow \mathbf{X} - \mathbf{T}^{(k)}(\mathbf{P}^{(k)})^T. \tag{3}$$

The convergence test consists in comparing two successive estimates of the eigenvalue λ and λ'. If the absolute difference $|\lambda' - \lambda|$ is smaller than some small error ε, then the convergence is achieved and the algorithm proceeds to the deflation step. Here, the maximum number of iterations is limited by some large enough J. Using the NIPALS-PCA algorithm approach, the decomposition of the data matrix $\mathbf{X}$ takes the form:

$$\mathbf{X} = \mathbf{T}_{(K)}\mathbf{P}_{(K)}^T + \mathbf{R}, \tag{4}$$

where $\mathbf{T}_{(K)} = [\mathbf{T}^{(0)}| \ldots |\mathbf{T}^{(K-1)}]$ is the matrix formed using the first K scores, $\mathbf{P}_{(K)} = [\mathbf{P}^{(0)}| \ldots |\mathbf{P}^{(K-1)}]$ is the matrix of the first K loadings, and $\mathbf{R}$ is the residual matrix. The pseudo-code of the NIPALS-PCA algorithm is given below:

$\mathbf{R} \leftarrow \mathbf{X}$
for $(k = 0, \ldots, K-1)$ do
    {
    $\lambda = 0$
    $\mathbf{T}^{(k)} \leftarrow \mathbf{R}^{(k)}$
    for $(j = 0, \ldots, J)$ do
        {
        $\mathbf{P}^{(k)} \leftarrow \mathbf{R}^T\mathbf{T}^{(k)}$
        $\mathbf{P}^{(k)} \leftarrow \mathbf{P}^{(k)}\|\mathbf{P}^{(k)}\|^{-1}$
        $\mathbf{T}^{(k)} \leftarrow \mathbf{R}\mathbf{P}^{(k)}$
        $\lambda' \leftarrow \|\mathbf{T}^{(k)}\|$
        if$(|\lambda' - \lambda| \le \varepsilon)$ then break
        $\lambda \leftarrow \lambda'$
        }
    $\mathbf{R} \leftarrow \mathbf{R} - \mathbf{T}^{(k)}(\mathbf{P}^{(k)})^T$
    }
return $\mathbf{T}$, $\mathbf{P}$, $\mathbf{R}$

2.3. GS-PCA algorithm

A well-known shortcoming of the NIPALS-PCA algorithm is the loss of orthogonality (Kramer, 1998). Both the computed scores $\mathbf{T}^{(k)}$ and the loadings $\mathbf{P}^{(k)}$, are supposed to be orthogonal. However, because of the errors accumulated at each iteration step (which involves large matrix-vector operations), this orthogonality is quickly lost, and in practice, one can compute accurately only the first few components. In order to stabilize the iterative PCA computation, from the orthogonality point of view, we propose an algorithm based on the Gram-Schmidt (GS) re-orthogonalization process. The classical GS algorithm (CGS) recursively constructs a set of orthonormal basis vectors for the subspace spanned by a given set of linearly independent normalized vectors (Golub et al., 1996). It is well known that the CGS algorithm is also numerically unstable due to rounding errors. However, the CGS can be easily stabilized by a small modification obtaining the so-called modified Gram-Schmidt (MGS) algorithm (Bjorck et al., 1992). Unfortunately, the MGS algorithm cannot be expressed by Level-2 BLAS functions (matrix-vector operations), and therefore it requires a substantial amount of global communications, when implemented on a parallel computer (Lingen, 2000). In contrast, the CGS algorithm can be easily expressed using matrix-vector operations, and therefore it is more suitable for parallel implementation. Also, the numerical stability of CGS can be achieved by applying it iteratively (Lingen, 2000). In the proposed GS-PCA algorithm, the re-orthogonalization correction is applied to both the scores and the loadings at each iteration step.

For the pseudo-code formulation of the GS-PCA algorithm, we prefer to use the truncated SVD description, since for $K = N$, the algorithm also returns the full SVD decomposition of the input matrix:

$$\mathbf{X} = \mathbf{V}_{(K)}\mathbf{\Lambda}_{(K)}\mathbf{U}_{(K)}^T + \mathbf{R} \tag{5}$$

where $\mathbf{V}_{(K)}$ and $\mathbf{U}_{(K)}$ are, respectively, the first K left and right eigenvectors, $\mathbf{\Lambda}_{(K)}$ are the corresponding eigenvalues, and $\mathbf{R}$ is the residual. One can easily show that $\mathbf{T}_{(k)} = \mathbf{V}_{(K)}\mathbf{\Lambda}_{(K)}$ and $\mathbf{P}_{(K)}^T = \mathbf{U}_{(K)}^T$. The pseudo-code of the GS-PCA algorithm can be formulated as following:

$\mathbf{R} \leftarrow \mathbf{X}$
for $(k = 0, \ldots, K-1)$ do
    {
    $\mu = 0$
    $\mathbf{V}^{(k)} \leftarrow \mathbf{R}^{(k)}$
    for $(j = 0, \ldots, J)$ do
        {
        $\mathbf{U}^{(k)} \leftarrow \mathbf{R}^T\mathbf{V}^{(k)}$
        if$(k > 0)$ then

```
            {
            A ← U_(k)^T U^(k)
            U^(k) ← U^(k) − U_(k) A
            }
        U^(k) ← U^(k) ||U^(k)||^-1
        V^(k) ← R U^(k)
        if(k ≥ 0) then
            {
            B ← V_(k)^T V^(k)
            V^(k) ← V^(k) − V_(k) B
            }
        λ_k ← ||V^(k)||
        V^(k) ← V^(k)/λ_k
        if(|λ_k − μ| ≤ ε) then break
        μ ← λ_k
        }
    R ← R − λ_k V^(k) (U^(k))^T
  }
T ← VΛ
P ← U
return T, P, R (for PCA) or V, U, Λ (for SVD)
```

One can see that, in every iteration step, if $k > 0$, then both the right (loads) and the left (scores) eigenvectors are re-orthonormalized. This procedure stabilizes the algorithm, but it also increases by a little bit the computational effort. However, this effort will be compensated by the efficiency of the parallel implementation. The GS-PCA algorithm assures the perfect orthogonality of both the loads and the scores. The errors accumulated in GS-PCA are only due to to the desired precision ε in the estimation of the eigenvalues λ_k. Also, for $K = N$ the GS-PCA algorithm returns a full SVD decomposition of the original matrix $\mathbf{X}$, with a maximum error ε for eigenvalues and perfectly orthogonal left/right eigenvectors.

3. IMPLEMENTATION DETAILS

The newly developed GPUs now include fully programmable processing units that follow a stream programming model and support vectorized single and double precision floating-point operations. For example, the CUDA computing environment provides a standard C like language interface to the NVIDIA GPUs (NVIDIA, 2008). The computation is distributed into sequential grids, which are organized as a set of thread blocks. The thread blocks are batches of threads that execute together, sharing local memories and synchronizing at specified barriers. An enormous number of blocks, each containing maximum 512 threads, can be launched in parallel in the grid.

In our implementation of NIPALS-PCA and GS-PCA algorithms, we use CUBLAS, a recent parallel implementation of BLAS, developed by NVIDIA on top of the CUDA programming environment (NVIDIA, 2008). CUBLAS library provides functions for:

- creating and destroying matrix and vector objects in GPU memory;
- transferring data from CPU mainmemory to GPU memory;
- executing BLAS on the GPU;
- transferring data from GPU memory back to the CPU mainmemory.

BLAS defines a set of low-level fundamental operations on vectors and matrices which can be used to create optimized higher-level linear algebra functionality. Highly efficient implementations of BLAS exist for most current computer architectures and the specification of BLAS is widely adopted in the development of high quality linear algebra software, such as the GNU Scientific Library (GSL) (Galassi et al, 2006). We have selected GSL CBLAS, for our host (CPU) implementation, due to its portability on various platforms (Windows/Linux/OSX, Intel/AMD), and because it is free and easy to use in combination with GCC (GNU Compiler). The GSL library provides a low-level layer which corresponds directly to the C-language BLAS standard, referred to here as CBLAS, and a higher-level interface for operations on GSL vectors and matrices.

The CBLAS (GNU Scientific Library) and, respectively, CUBLAS (NVIDIA CUDA) implementations of the NIPALS-PCA and GS-PCA algorithms require the following Level 1, 2, and 3 BLAS functions (see the CBLAS/CUBLAS programming manuals for definition details [NVIDIA, 2008]:

CBLAS (Level 2): gsl_blas_dgemv
CUBLAS (Level 2): cublasDgemv

- computes in double precision the matrix-vector product and sum:

$$\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta \quad or \quad \mathbf{y} \leftarrow \alpha\mathbf{A}^T\mathbf{x} + \beta \tag{6}$$

α and β are double precision scalars, and $\mathbf{x}$ and $\mathbf{y}$ are double precision vectors. $\mathbf{A}$ is a matrix consisting of double precision elements. Matrix $\mathbf{A}$ is stored in column major format.

CBLAS (Level 1): gsl_blas_daxpy
CUBLAS (Level 1): cublasDaxpy

- computes the double precision sum:

$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}, \tag{7}$$

multiplies double precision vector $\mathbf{x}$ by double precision scalar α and adds the result to double precision vector $\mathbf{y}$.

CBLAS (Level 1): gsl_blas_dnrm2
CUBLAS (Level 1): cublasDnrm2

- computes the Euclidean norm of a double precision vector $\mathbf{x}$:

$$\|\mathbf{x}\|_2 \leftarrow \sqrt{\sum_{m=0}^{M} x_m^2}. \tag{8}$$

CBLAS (Level 3): gsl_blas_dger
CUBLAS (Level 3): cublasDger

- computes in double precision the matrix-matrix sum:

$$\mathbf{A} \leftarrow \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}, \tag{9}$$

where α is a double precision scalar, $\mathbf{x}$ is an M element double precision vector, $\mathbf{y}$ is an N element double precision vector, and $\mathbf{A}$ is an $M \times N$ matrix consisting of double precision elements. Matrix $\mathbf{A}$ is stored in column major format.

These are the critical functions/kernels that are efficiently exploited in the parallel CUBLAS implementation. The other involved functions are for vector/matrix memory allocation and vector/matrix accessing, device (GPU) initialization, host-device data transfer and error handling.

In the CUBLAS implementation, the data space is allocated both on host (CPU) mainmemory and on device (GPU) memory. After the data is initialized on host it is transferred on device, where the main parallel computation occurs. The results are then transferred back on host memory.

The source code of the double precision CBLAS and CUBLAS implementations is available online at *xxx.lanl.gov/abs/0811.1081*. These implementations can be easily modified in order to meet the end user's specifications.

4. RESULTS AND DISCUSSION

The numerical tests have been carried out on the following system: AMD Phenom 9950 CPU (2.6 GHz); XFX GTX280 GPU; NVIDIA Linux 64-bit driver (177.67); CUDA Toolkit and SDK 2.0; Ubuntu Linux 64-bit 8.04.1, GNU Scientific Library v.1.11; Compilers: GCC (GNU), NVCC (NVIDIA). The GPU used is a high-end graphics card solution with 240 stream processors and 1×Gb DDR3 RAM, which supports both single and double precision, and it is theoretically capable of 1-Tflop computational power.

In Figure 1, we give the CPU versus GPU execution time as a function of the size of the randomly generated input matrix $\mathbf{X}(M \in [5\times10^2, 1.5\times10^4], N = M/2, K = 10, \varepsilon = 10^{-7})$. The time gap between

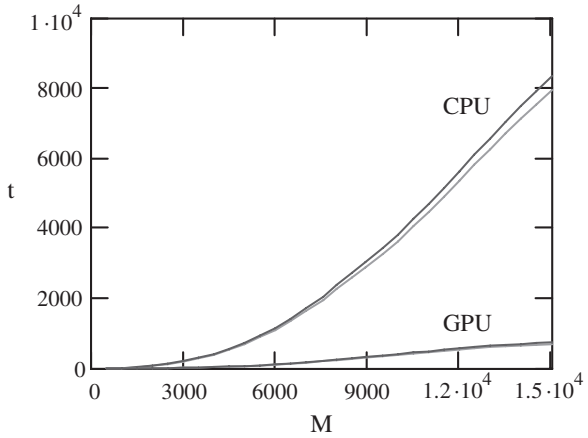

FIG. 1. GPU versus CPU execution time of the NIPALS-PCA and GS-PCA algorithms as a function of the size of the input matrix $\mathbf{X}(M, N = M/2, K = 10)$ (Gray = NIPALS-PCA; Black = GS-PCA).

CPU and GPU increases very fast by increasing the size of the input matrix, and the CPU time versus the GPU time reaches a maximum for $M = 1.5 \times 10^4$, where the GPU is about 12 times faster than the CPU. The GS-PCA algorithm is only about 5–7% slower than the standard NIPLALS-PCA algorithm, in both CPU and GPU implementation. These results also show that the GPU performance is dependent on the scale of the problem. Thus, in order to exploit efficiently the massive parallelism of GPUs and to effectively use the hardware capabilities, the problem itself needs to scale accordingly, such that thousands of threads are defined and used in computation.

In conclusion, we have presented an iterative GS-PCA algorithm based on Gram-Schmidt re-orthogonalization. The GS-PCA algorithm assures the perfect orthogonality of both the loads and the scores, and thus totally eliminates the loss of orthogonality present in the standard NIPALS-PCA algorithm. Also, we have discussed the GPU parallel implementation of both NIPALS-PCA and GS-PCA algorithms. We have shown that the GPU parallel optimized versions, based on CUBLAS (NVIDIA), are substantially faster (up to 12 times) than the CPU-optimized versions based on CBLAS (GNU Scientific Library).

ACKNOWLEDGMENTS

I acknowledge the financial support from IBI and the University of Calgary.

DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Bjorck, A., and Paige, C. 1992. Loss and recapture of orthogonality in the modified Gram-Schmidt algorithm. *SIAM J. Matrix Anal. Appl.* 13, 176–190.

Galassi, M., Davies, J., Theiler, J., et al. 2006. *GNU Scientific Library Reference Manual*, 2nd ed. rev. Network Theory Limited.

Golub, G.H. 1996. *Matrix Computations*, 3rd ed. John Hopkins University Press, Baltimore.

Jackson, J.E. 1991. *A User's Guide to Principal Components*. John Wiley and Sons, New York.

Jolliffe, I.T. 2002. *Principal Component Analysis*. Springer, New York.

Kramer, R. 1998. *Chemometric Techniques for Quantitative Analysis*. CRC Press, New York.

Lingen, F.J. 2000. Efficient Gram-Schmidt orthonormalisation on parallel computers. *Commun. Numer. Methods Eng.* 16, 57–66.

NVIDIA. 2008. *CUDA Compute Unified Device Architecture*. *Programming Guide*. CUBLAS Library.

Wold, S., Esbensen, K., and Geladi, P. 1987. Principal component analysis. *Chemometrics Intellig. Lab. Syst.* 2, 37–52.

Address correspondence to:
Dr. Mircea Andrecut
Institute for Biocomplexity and Informatics
University of Calgary
2500 University Drive NW
Calgary, Alberta, T2N 1N4, Canada

E-mail: mandrecu@ucalgary.ca