Successive Rank-Revealing Cholesky Factorizations on GPUs

Ty Fridrich, Nikos Pitsianis and Xiaobai Sun
Department of Computer Science, Duke University, Durham, NC 27708
{Ty,Nikos,Xiaobai}@CS.Duke.edu

Introduction

We present an algorithm and its GPU implementation for fast generation of rank-revealing Cholesky factors $\{R_k\}$ at output in response to a sequence of data matrices $\{A_k\}$ at input. The Cholesky factors are subsequently used for calculating adaptive weight vectors as control feedback in space-time adaptive processing (STAP) and sensing systems [3]. The size of the input data matrices is $m \times n$, where n is the number of sensors, and m is the number of samples or snapshots. Usually, $m = k \cdot n$ with k between 3 and 5. In [1, 4] we introduced an algorithm for accelerating successive Cholesky factorizations and a GPU implementation. The algorithm exploits the redundancy in the data matrices with a new, un-conventional approach. It is advantageous in comparison to conventional adaptation algorithms in arithmetic operation complexity, memory space and the concurrency for parallel implementation within each factorization and between consecutive factorizations. Here, the algorithm is extended to the case where the resulting Cholesky factors have rank-revealing structures. It appears the first effective and efficient algorithm for such a computational task. The other approaches either ignore the data redundancy at the cost of more arithmetic operations or tackle the redundancy with highly sequential means. The GPU implementation underscores the algorithmic concurrency and represents an additional mapping from architecture-independent concurrency to architecture-specific parallel computation.

The algorithm

The algorithm exploits the data redundancy among successive matrices. For each $k \geq 1$, the matrix A_{k+1} can be viewed as the forward shift of A_k with the last row containing the most recent data. This relationship may be formally described as follows: $A_{k+1} = S\, A_k + e_m\, (d_{new} - a_1)^T$, where S is the circulant up-shifting matrix, $e_m=(0,\ldots,0,1)^T$ in R^m, d_{new} is the new data to be placed as the last row of A_{k+1}, and a_1 is the first row of A_k to be discarded. A simple way is to ignore the relationship and apply a rank-revealing factorization algorithm to each and every matrix [2]. Without the requirement on rank-revealing, one may apply a typical conventional rank-1 update algorithm [2], which is highly sequential within each update and between successive updates. Such kind of rank-1 update algorithm can not be applied easily and efficiently to the case with rank-revealing requirement. In contrast, the concurrent adaptation approach we introduced in [1, 4] exploits the large portion in each matrix that is common with its neighboring matrices, not limited to one neighbor. Moreover, it can accommodate the rank-revealing treatment effectively and efficiently.

Acknowledgments. This project is supported in part by the MTO program of DARPA.

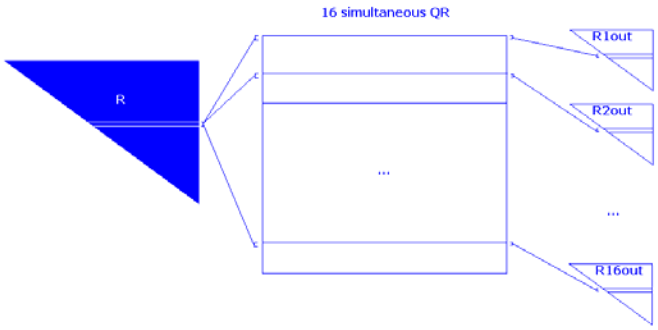

Figure 1: Parallel adaptation

The algorithm treats every p matrices as a *bloc*, where p is the bloc size, $1<p<m$. Denote p consecutive neighbor matrices by $A_1, A_2, \ldots, A_p$. There are m-p+1 rows common to all the matrices. These rows can be pre-determined as a sub-matrix of the first matrix, $A_c=A_1(p:m,1:n)$, without the knowledge of the new data in the subsequent matrices. Here we use MATLAB notation for matrix partition and composition. There are two basic steps for factorizing the matrices in each bloc. First, apply a rank-revealing factorization procedure to A_c and obtain a rank-revealing Cholesky factor R_c and an associated permutation matrix P_c. Namely, $A_cP_c=Q_cR_c$, for some matrix Q_c with orthonormal columns, which is explicitly formed, where R_c may be seen as a 2-by-2 block triangular matrix, with the leading block $R_c(1:r,1:r)$ corresponding to the r-dimensional numerical column space of A_c, and the tailing block to the numerical null space.

In the second step, we obtain the rank-revealing factors R_k, $k=1:p$, by adapting the common factors of A_c to each and every individual matrix in the bloc. Specifically, the rank-revealing Cholesky factor R_1, of A_1, can be obtained quickly by completing the factorization of the matrix $[A_1\ (1:p-1,1:n)\ P_c;\ R_c]P_1=U_1R_1$, where U_1 has orthonormal columns. One shall notice that the permutation matrix P_1 does not necessarily involve the first r columns of the trapezoid matrix $[A_1\ (1:p-1,1:n)\ P_c;\ R_c]$, which are already numerically linearly independent. In other words, the numerical null space of the trapezoid matrix is in the subspace spanned by the trailing columns. Thus, the adaptation in the first r columns can be done as efficiently as that without rank-revealing treatment, i.e., as any Q-less QR factorization can be. The dimension of the null space is

usually small in comparison to r, the dimension of the rank space. The first step has narrowed the search of the gap between the null and rank spaces to the null space corresponding to the trailing block of R_c. The computation of the first Cholesky factor is not compromised in accuracy and arithmetic complexity, nor delayed temporally.

Similarly, the next Cholesky factor R_2 is obtained from the semi-processed matrix $[A_1(2:p-1,1:n)\ P_c;\ A_2(m,1:n)\ P_c,\ R_c]$. This implies a significant reduction in arithmetic operations than that by re-starting the factorization process at A_2. In addition, the computation of R_2 can be started as soon as the new data $A_2(m,1:n)$ is available, with no need of waiting for the first factor to complete. In general, R_k is obtained from matrix $[A_k(m:-1:m-k+2,1:n)\ P_c;\ A_1(k:p-1,1:n)\ P_c,\ R_c]$. The adaptation of R_c to R_k can start no later than the data in the last row of A_k is made available. In terms of arithmetic operation complexity, the optimal size of the common block is determined by $p=\sqrt{m}$ approximately. In this case, the number of arithmetic operations per factorization is reduced to $\gamma\sqrt{m}\ n^2$ with a modest constant γ. The memory space can be restricted to $(m+\sqrt{m})(n+3)$, with at most 2m memory units for the information of the current orthogonal transformation and m memory units for the permutations. Depending on detailed implementation arrangement, the data redundancy may be further exploited in the adaptation step as well between p/2 neighbors.

Parallelization

The new algorithm introduces the concurrency in each adaptive factorization to nearly the same level as that for QR factorization, better than that for rank-revealing QR factorization, when the null space is of low dimension. It introduces also the concurrency between successive factorizations. Similar to the standard QR factorization, the successive Cholesky factorizations, as described above, use two elementary orthogonal transforms, Givens rotations and Householder reflections. These may be viewed as the two extremes in the granularity of employing orthogonal transformations. Moreover, for each version of the successive factorization algorithm, a GPU implementation may differ from one to another in data layout, data partition, operation partition and operation scheduling.

Consider first the computation of every p consecutive Cholesky factors with Givens rotations only. There are many different ways to order Givens rotations for each factorization. But many orderings suitable for Cholesky factorizations are not suitable for rank-revealing Cholesky factorizations. The computation of the common Cholesky factor R_c amounts to different orderings in the Givens rotations for each individual matrix. This common factorization step saves $p-1$ similar factorizations. Consider next the factorization with Householder reflections. The computation of the common factor R_c entails an additional partition in the size of the Householder transformations for each individual matrix, in comparison to the very basic QR factorization using Householder reflections~\cite{GvanL:1989}. The partition location shifts from one matrix to the next. For R_1, the additional partition requires few more arithmetic operations than that with the basic Household-QR version. However, it saves p-1 factorizations over the same sub-matrix in the next p-1 matrices.

With any selection or combination of the orthogonal transforms, the computations in adapting R_c to p consecutive Cholesky factors are concurrent when all the data are available. In other words, the adaptation step does not impose any additional sequential dependence between successive factorizations as a conventional update algorithm does. Within each adaptive factorization, the parallelism is almost the same as any QR factorization without rank revealing.

In addition to the ordering in data arrival, an implementation of the algorithm must also respect the availability, capacity and constrains on a specific architecture system, including both hardware and software components. We demonstrate with a particular implementation of the algorithm for successive Cholesky factorizations using Householder reflections and employing GPU-friendly parallelization techniques. We present a graphical description of a parallel implementation of the adaptation step, with special consideration to the present and near-future GPU architectures. In particular, we keep the spatial redundancy in the parallel adaptation step as low as possible. This is to increase the degree of parallelism within a given memory space, increase the locality of memory access, and reduce the data movement between memory and cache. The counterpart version using Givens rotations is under development.

References

[1] T. Fridrich, N. P. Pitsianis, and X. Sun, "A GPU implementation of successive Cholesky factorizations," In *Workshop on Edge Computing Using New Commodity Architectures*, pp D--51:52, Chapel Hill, NC, May 2006.

[2] G. Golub and C. F. Van Loan, *Matrix Computations*, 1989.

[3] W. L. Melvin, "A STAP overview," *IEEE A & E Sys. Magazine*, Vol. 19, No. 2, pp 19-35, 2004.

[4] X. Sun, "Accelerated generation of Cholesky factor sequence in space-time adaptive processing," *TR 2006-06, Duke University, Department of Computer Science, May 2006.*