J Supercomput (2011) 58:429–437
DOI 10.1007/s11227-011-0606-4

Using graphics processors to accelerate the computation of the matrix inverse

P. Ezzatti · E.S. Quintana-Ortí · A. Remón

Published online: 15 April 2011

Abstract We study the use of massively parallel architectures for computing a matrix inverse. Two different algorithms are reviewed, the traditional approach based on Gaussian elimination and the Gauss–Jordan elimination alternative, and several high performance implementations are presented and evaluated. The target architecture is a current general-purpose multicore processor (CPU) connected to a graphics processor (GPU). Numerical experiments show the efficiency attained by the proposed implementations and how the computation of large-scale inverses, which only a few years ago would have required a distributed-memory cluster, take only a few minutes on a hybrid architecture formed by a multicore CPU and a GPU.

Keywords Linear algebra · Matrix inversion · Graphics processors

1 Introduction

Matrix inversion appears in a few scientific applications of different areas (e.g., model reduction, polar decomposition, optimal control, prediction, etc.) and requires an important computational effort in terms of execution time and memory. Thus, matrix inversion is a suitable operation for new highly parallel architectures, like GPUs or general-purpose multicore processors.

P. Ezzatti
Centro de Cálculo—Instituto de Computación, Universidad de la República, 11.300 Montevideo, Uruguay
e-mail: pezzatti@fing.edu.uy

E.S. Quintana-Ortí · A. Remón (✉)
Dept. de Ingeniería y Ciencia de Computadores, Universidad Jaime I, 12.071 Castellón, Spain
e-mail: remon@icc.uji.es

E.S. Quintana-Ortí
e-mail: quintana@icc.uji.es

In this paper, we evaluate a variety of high performance implementations for matrix inversion that exploit all the computational capabilities offered by an hybrid architecture formed by a multicore CPU and a GPU. The study includes the revision of two methods for the computation of a matrix inverse and several high-performance implementations for each method. The numerical experiments illustrate the efficiency attained by the Gauss–Jordan elimination implementations on the target architecture.

The rest of the paper is structured as follows. In Sects. 2 and 3, we describe different algorithms and implementations for matrix inversion. This is followed by experimental results in Sect. 4. Finally, in Sect. 5, a few concluding remarks and open questions are exposed.

2 High-performance matrix inversion

This section presents two strategies to compute the inverse of a general unsymmetric matrix, the traditional technique based on Gaussian elimination (i.e., the LU factorization) and the Gauss–Jordan elimination method.

2.1 Matrix inversion via the LU factorization

The traditional approach to compute the inverse of a matrix $A \in \mathbb{R}^{n\times n}$ is based on the LU factorization, and consist of the following four steps:

1. Compute the LU factorization $PA = LU$, where $P \in \mathbb{R}^{n\times n}$ is a permutation matrix, and $L, U \in \mathbb{R}^{n\times n}$ are, respectively, unit lower and upper triangular factors [6].
2. Invert the triangular factor $U \rightarrow U^{-1}$.
3. Solve the lower triangular system $XL = U^{-1}$ for X.
4. Undo the permutations $A^{-1} := XP$.

The computational cost of computing a matrix inversion following the previous four steps is $2n^3$ flops (floating-point arithmetic operations).

2.2 Matrix inversion via the Gauss–Jordan elimination

The Gauss–Jordan elimination algorithm [5] (GJE) for matrix inversion is, in essence, a reordering of the computations performed by matrix inversion methods based on Gaussian elimination, and hence requires the same arithmetic cost.

Figure 1 illustrates a blocked version of the GJE procedure for matrix inversion using the FLAME notation [4, 7]. There $m(A)$ stands for the number of rows of matrix A. We believe the rest of the notation to be intuitive; for further details, see [4, 7]. A description of the unblocked version, called from inside the blocked one, can be found in [8]; for simplicity, we hide the application of pivoting during the factorization, but details can be found there as well.

The bulk of the computations in procedure GJE$_{\text{BLK}}$ can be cast in terms of the matrix-matrix product, an operation with a high parallelism. Therefore, GJE$_{\text{BLK}}$ is a highly appealing method for matrix inversion on emerging architectures like GPUs, where many computational units are available, especially if a tuned implementation of the matrix-matrix product is available.

```
Algorithm: $[A] := \mathrm{GJE}_{\mathrm{BLK}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $A_{BR}$ is $n \times n$

while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        where $A_{11}$ is $b \times b$
    ============================================================
    $\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} := \mathrm{GJE}_{\mathrm{UNB}}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$      Unblocked Gauss-Jordan
    $A_{00} := A_{00} + A_{01}A_{10}$                  Matrix-matrix product
    $A_{20} := A_{20} + A_{21}A_{10}$                  Matrix-matrix product
    $A_{10} := A_{11}A_{10}$                           Matrix-matrix product
    $A_{02} := A_{02} + A_{01}A_{12}$                  Matrix-matrix product
    $A_{22} := A_{22} + A_{21}A_{12}$                  Matrix-matrix product
    $A_{12} := A_{11}A_{12}$                           Matrix-matrix product
    ============================================================
    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

Fig. 1 Blocked algorithm for matrix inversion via GJE without pivoting

3 High-performance implementations

3.1 Implementations via the LU factorization

The algorithm presented in Sect. 2.1 is composed of four steps that have to be computed in that strict order. We can identify in this algorithm two drawbacks from the parallel computing viewpoint:

- The algorithm sweeps through the matrix four times (one time per step), so there are many repeated memory accesses.
- Operating with triangular factors may be a source of load imbalance.

LAPACK [2] is a high-performance linear algebra library, for general-purpose processors that comprises routines covering the functionality required by this algorithm. In particular, routine `getrf` obtains the LU factorization (with partial pivoting) of a general matrix (Step 1), while routine `getri` computes the inverse matrix of A using the LU factorization obtained by `getrf` (Steps 2–4).

3.1.1 Implementation on a multi-core CPU: LU(CPU)

MKL offers a multithreaded version of BLAS for multicore CPUs. The thread-level parallelism of BLAS carries over to the implementation of LAPACK in MKL. Thus, the simple use of the multithreaded implementation of MKL exploits the parallelism inside the BLAS calls performed from routines `getrf` and `getri`.

3.1.2 Implementation on a many-core GPU: LU(GPU)

For this implementation, we developed GPU versions of routines `getrf` and `getri`, as well as `getf2` and `trtri` which are invoked from the former two. All the codes extract fine-grained parallelism using specific BLAS kernels for the GPU (e.g., CUBLAS). This implementation also requires that the matrix is initially sent to the GPU and the inverse is retrieved from there once it has been computed.

3.1.3 Hybrid implementation: LU(MAGMA)

This is the implementation based on the use of computational kernels from the MAGMAlibrary. In this version, routines `getrf`, `trsm`, and `gemm` from MAGMA [1] are used to obtain the LU factorization of the matrix, solve the triangular system, and compute the product of matrices, respectively. As routine `trtri` needed to compute the inverse of a triangular matrix is not implemented in MAGMA, we have employed a high-performance implementation developed by researchers at AICES-RWTHD. As in the previous case, this variant also requires the initial transfer of data from CPU to GPU and the final communication of the result in the inverse direction.

3.2 Implementations via the Gauss-Jordan elimination

In this subsection, we describe four implementations for the GJE method (with partial pivoting) on the two parallel architectures (multicore CPU and a GPU from NVIDIA). The variants differ mainly on which parts of the computation are performed on the CPU (the general-purpose processor or host), and which operations are off-loaded to the hardware accelerator (the GPU or device). They all aim at reducing the number of communications between the memory spaces of the host and the device.

3.2.1 Implementation on a multi-core CPU: GJE(CPU)

In this implementation all operations are performed on the CPU. Parallelism is obtained from a multi-threaded implementation of BLAS. Since most of the computations are cast in terms of matrix-matrix products, high performance can be expected.

3.2.2 Implementation on a many-core GPU: GJE(GPU)

This is the GPU-analogue to the previous variant. The matrix is first transferred to the device; all computations are performed there, and finally the result (the matrix inverse) is retrieved to the host. Again, all the parallelism is extracted from a multi-threaded implementation of BLAS on a GPU(e.g., the implementation from NVIDIA, CUBLAS).

3.2.3 Hybrid implementation: GJE(Hybrid)

While most of the operations performed in the GJE algorithm are well suited for the GPU, a few are not. This is the case for fine-grained operations, where the low computational cost and data dependencies deliver little performance on massively parallel architectures like GPUs. To solve this problem, Benner et al. [3] proposed a hybrid version in which operations are performed in the most convenient device, exploiting the capabilities of both architectures.

In this variant, the matrix is initially transferred to the device, then the iterative algorithm in Fig. 1 is computed jointly by both architectures, and finally the inverse is moved back to the host. In particular, only the factorization of the current column panel, composed of $[A_{01}^T; A_{11}^T; A_{21}^T]^T$, is executed on the CPU, since it involves a reduced number of data (limited by the algorithmic block size), pivoting and level-1 BLAS operations which are not well suited for the architecture of the GPU. The matrix-matrix products and pivoting of the columns outside the current column panel are performed on the GPU using BLAS kernels (e.g., in the CUBLAS library).

3.2.4 Multilevel hybrid implementation: GJE(Hyb-ML)

Although GJE (Hybrid) attains an important computational efficiency due to the fact that each operation is executed on the most convenient architecture, all stages are performed sequentially. Variant GJE (Hyb-ML) targets the concurrent execution of operations in both architectures.

In order to achieve this, we apply some minor changes to obtain a look-ahead variant [9] of the algorithm in Fig. 1, that enables concurrent computations on CPU and GPU:

1. The first column panel ($[A_{01}^T; A_{11}^T; A_{21}^T]^T$) is factored on the CPU.
2. The active column panel is transferred to the GPU.
3. The first b columns of block $[A_{02}^T; A_{12}^T; A_{22}^T]^T$ (that is, block $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]^T$ of the next iteration) are updated and transferred to the CPU.
4. While the GPU update blocks $[A_{00}^T; A_{10}^T; A_{20}^T]^T$, and the remaining part of $[A_{02}^T; A_{12}^T; A_{22}^T]^T$, the CPU factorizes $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]^T$.
5. Move the factorization forward by b columns and repeat steps (b)–(d) until the full matrix inverse is computed.
6. All the GPUs transfer their corresponding column block to the host.

The efficiency of the look-ahead variant can be limited by the algorithmic block size (b). The optimal block size for the GPU is usually larger than that for the CPU; in other words, if we set b to the optimal block size for the GPU, we will slow down the execution on the CPU. Therefore, we introduce an additional modification to simultaneously optimize the efficiency in both architectures operating with two different block sizes (one for the CPU and one for the GPU); especially, as in GJE (Hybrid), a blocked implementation of the GJEmethod is applied to matrix A, but this time the CPU factorizes $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]^T$ using a blocked algorithm (i.e., GJE$_{\text{BLK}}$) instead of its unblocked version. Thus, the CPU executes algorithm GJE$_{\text{BLK}}$ on panel $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]^T$ using its optimal block size (b_c) while at a higher level, algorithm GJE$_{\text{BLK}}$ is executed with the optimal block size for the GPU.

Table 1 Hardware employed in the experiments

Platform	Processors	#cores	Frequency (GHz)	L2 cache (MB)	Memory (GB)
PECO	Intel Xeon QuadCore E5520	8	2.27	8	24
	Nvidia TESLA c1060	240	1.3	–	4
ZAPE	AMD Phenom QuadCore 9550	4	2.20	0.5	4
	Nvidia GTX 480	480	1.4	–	1.5

4 Experimental results

In this section, we evaluate the parallel implementations described in Sect. 3 for the computation of a matrix inverse.

Two target platforms consisting of a multicore CPU connected to a GPU have been tested; see Table 1. The first platform, PECO, consists of two Intel Xeon E5520 (Nehalem) QuadCore processors at 2.27 GHz connected to an NVIDIA Tesla C1060 GPU. The second platform, ZAPE, consists of an AMD 9550 (Phenom) QuadCore at 2.2 GHz connected to an NVIDIA GTX480 (Fermi) GPU. Notice that, in general, the CPU in PECO is faster than that in ZAPE, but the GPU in the latter platform outperforms that of PECO. Intel MKL 10.1 implementation of BLAS and LAPACK is employed to compute most of the operations on the general purpose processors, while the NVIDIA CUBLAS (version 2.1 for PECO; 3.0.14 for ZAPE) and MAGMA (version 0.2) libraries are employed on the GPUs.

We set OMP_NUM_THREADS to the number of cores on the CPU, so that one thread is employed per core in the parallel execution of the MKL routines.

The different implementations of matrix inversion were evaluated for a variety of matrix dimensions ($n = 1000$–14000 on PECO and $n = 1000$–13000 on ZAPE). For each matrix dimension, several block sizes ($b = 32, 64, 96, 128, 256, 384, 512$) were tested. In the case of the GJE (Hyb-ML) implementation, three values for the CPU block size ($b_c = 8,16, 32$) were employed. For simplicity, only the results obtained with the optimal pair (b, b_c) are reported.

All experiments employ single precision floating-point arithmetic, and the results include the communication times between the host and the device memory spaces. Performance is reported in terms of GFLOPS (10^9 flops/s)

Figure 2 shows the results for PECO. Implementations based on the Gauss–Jordan elimination are very efficient, especially for the inversion of large matrices. The highly tuned implementation from LAPACK is clearly the best option for small matrices, but it is also the slowest for medium/large matrices. Due to the large number of cores, the GPU implementations are very efficient on the inversion of large matrices, but the time required by the CPU-GPU transfers makes them inefficient for small matrices (e.g., for the GJE (GPU) version, this approximately represents 15% of the total time for matrices of dimension 1024, but less than 4% for matrices of dimension 14016). Hybrid approaches (LU (MAGMA) and GJE (Hyb-ML)) obtain the best results for matrices with $n > 2000$. They exploit the capabilities of the underlying platform and, simultaneously, keep under control the overhead introduced by communications. Both implementations obtain similar results for small/medium matrices, but variant GJE (Hyb-ML) is faster for the inversion of large matrices.

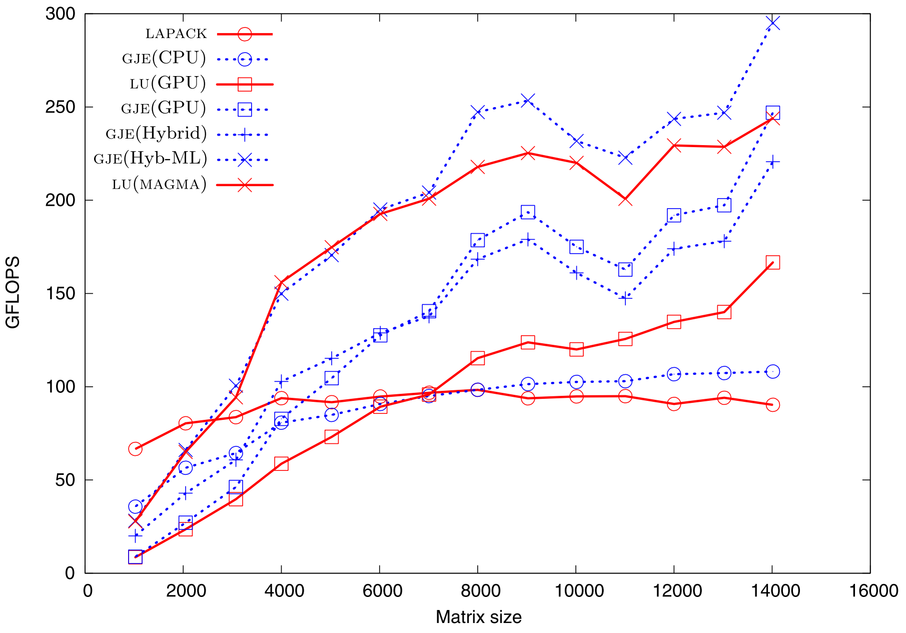

Fig. 2 Performance (GFLOPS) attained by matrix inversion on PECO

Figure 3 shows the results obtained on ZAPE. Again the hybrid implementations obtain the best results, and the GJE-based algorithms clearly outperform the alternatives based on Gaussian elimination. The gap between the GPU and the CPU capabilities in ZAPE is larger than in PECO, as:

- Hybrid and GPU implementations are faster than CPU implementations even for small matrices.
- The best implementation is 10× faster than LAPACK, while in PECO the best implementation is only 3× faster.
- The performance partially stabilizes for large matrices, because it is limited by the highest performance offered by the `gemm` routine in CUBLAS.

In summary, the GJE (Hyb-ML) implementation clearly outperforms the rest, except for small matrices on PECO, where LAPACK attains the best performance. Thus, we can conclude that variant GJE (Hyb-ML) can be easily adapted to the target platform and the specific matrix dimension, providing high performance for all the CPU-GPU platforms and matrix sizes.

5 Concluding remarks

This paper reviews inversion of general large-scale matrices on hybrid CPU-GPU platforms. Two methods and several implementations are presented and evaluated, resulting in the following conclusions:

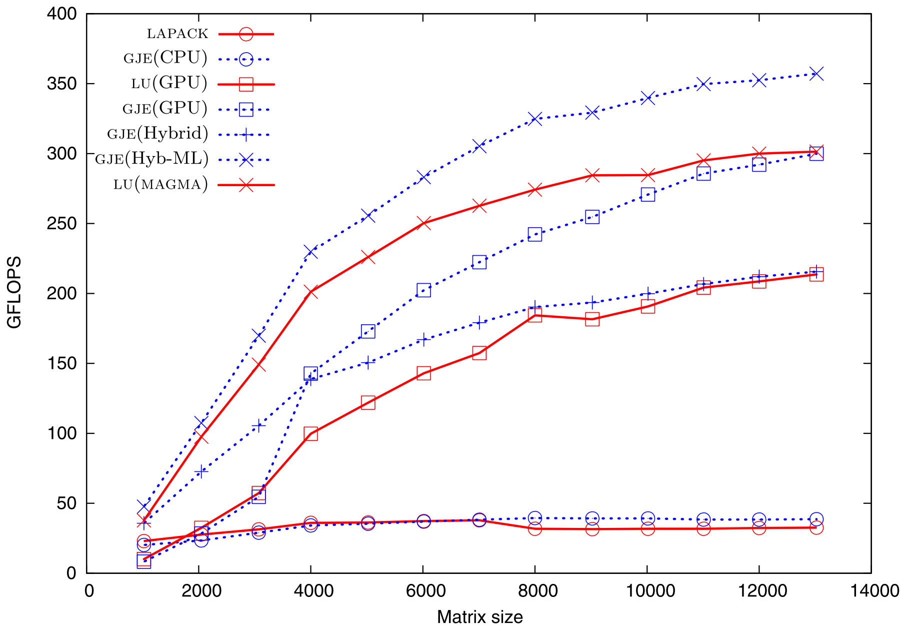

Fig. 3 Performance (GFLOPS) attained by matrix inversion on ZAPE

- The GJE method is a well-suited procedure for parallel computing, reaching better performance than the traditional Gaussian elimination approach.
- Hybrid implementations can efficiently exploit the underlying platform features, and perform well for small and large matrices.
- The GPU implementations are efficient for large matrices, but inefficient for small matrices due to the communication overhead.
- The GJE (Hyb-ML) implementation obtains high performance for all the matrix dimensions and platforms evaluated.

The study has introduced some questions that should be addressed in the future.

- Double precision arithmetic is required in some applications, but pose some challenges to our implementations.
  - The cost of data transfers is higher.
  - The gap between the CPU and GPU performance is reduced even for the late generation of GPUs.
- An automatic procedure to obtain the optimal block size for a given matrix and platform can significantly decrease the evaluation time.
- Platforms with multiple CPUs and GPUs should be addressed.

Acknowledgements We thank D. Fabregat and P. Bientinesi from the *Aachen Institute for Advanced Study in Computational Engineering Sciences (AICES)* from the *RWTH Aachen University*, for their collaboration and their high performance routines for the triangular matrix inversion on GPUs. This work was partially supported by the Spanish Ministry of Science and Innovation/FEDER (contract TIN2008-06570-C04-01).

References

1. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J Phys Conf Ser 180(1):012037
2. Anderson E, Bai Z, Bischof C, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide, 3rd edn. SIAM, Philadelphia
3. Benner P, Ezzatti P, Quintana ES, Remón A (2009) Using hybrid cpu-gpu platforms to accelerate the computation of the matrix sign function. In: LNCS, 7th int workshop on algorithms, models and tools for parallel computing on heterogeneous networks. Springer, Berlin
4. Bientinesi P, Gunnels JA, Myers ME, Quintana-Ortí ES, van de Geijn RA (2005) The science of deriving dense linear algebra algorithms. ACM Trans Math Softw 31(1):1–26
5. Gerbessiotis AV (1997) Algorithmic and practical considerations for dense matrix computations on the BSP model. PRG-TR 32, Oxford University Computing Laboratory
6. Golub G, Loan CV (1996) Matrix computations, 3rd edn. The Johns Hopkins University Press, Baltimore
7. Gunnels JA, Gustavson FG, Henry GM, van de Geijn RA (2001) FLAME: formal linear algebra methods environment. ACM Trans Math Softw 27(4):422–455
8. Quintana-Ortí E, Quintana-Ortí G, Sun X, van de Geijn R (2001) A note on parallel matrix inversion. SIAM J Sci Comput 22:1762–1771
9. Strazdins A (1998) A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. TR-CS-98-07 07, The Australian National University