Improving the performance of the matrix inversion on a Tesla GPU

Pablo Ezzatti[1], Enrique S. Quintana-Ortí[2], and Alfredo Remón[2]

[1] Centro de Cálculo–Instituto de la Computación, Universidad de la República, 11.300–Montevideo, Uruguay, pezzatti@fing.edu.uy
[2] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain, {quintana,remon}@icc.uji.es

Abstract. We study two different techniques for the computation of a matrix inverse, the traditional approach based on Gaussian factorization and the Gauss-Jordan elimination alternative more suitable for parallel architectures. The target architecture is a current general-purpose multi-core processor (CPU) connected to a graphics processor (GPU). Parallelism is obtained from the use of libraries MKL (for the CPU) and CUBLAS (for the GPU), as well as, performing simultaneously operations in both architectures. Numerical experiments performed on a system equipped with two Intel QuadCore processors and a Tesla C1060 GPU, illustrate the efficiency attained by the Gauss-Jordan elimination implementation.

1 Introduction

Matrix inversion appears in a few scientific applications of different areas and requires an important computational effort.

Due to the amount of data and the number of floating point operations needed ($O(n^3)$ floating-point arithmetic operations, where n is the matrix dimension), matrix inversion is a suitable operation for new highly parallel architectures, like GPUs or multi-core general purpose processors.

In this paper we evaluate high performance implementations for matrix inversion on a hybrid CPU-GPU platform. Parallelism is extracted from the use of parallel libraries (a multi-threaded version of BLAS for the CPU, and CUBLAS for the GPU) and also from the concurrent execution of operations in both architectures.

The numerical experiments presented demonstrate that large-scale problems which, only a few years ago, would have required a distributed-memory cluster, can now be solved on a hybrid architecture formed by a CPU and a GPU.

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

2 High-Performance Matrix Inversion

This section presents two strategies to obtain a matrix inverse, the traditional technique based on Gaussian elimination and the Gauss-Jordan elimination method. Also, some implementations for each one of those techniques are decribed.

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 Gaussian elimination (i.e., 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 [5].
2. Invert the triangular factor $U \rightarrow U^{-1}$.
3. Solve the lower triangular system $XL = U^{-1}$ for X.
4. Undo the permutations $A^{-1} := XP$.

LAPACK [1] is a high-performance linear algebra library which provides routines that cover the functionality required in the previous steps. In particular, routine `getrf` obtains the LU factorization (with partial pivoting) of a nonsingular matrix (Step 1), while routine `getri` computes the inverse matrix of A using the LU factorization obtained by `getrf` (Steps 2–4).

The computational cost of computing a matrix inversion following the previous four steps is $2n^3$ flops (floating-point arithmetic operations). The algorithm sweeps through the matrix four times (one time per step) and presents a poor load balance, due to the work with the triangular factors.

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

We use the MKL library (from Intel Corporation), which is an implementation of LAPACK. MKL offers a multi-threaded version for multi-core CPUs.

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

For this implementation, we developed GPU versions of routines `getrf` and `getri`. Since `getf2` and `trtri` routines were also required, it was necessary to implement GPU versions of them too. All the developed codes are based on the use of BLAS kernels (e.g. CUBLAS). The execution steps of this implementation are: send the matrix from the host to the GPU, compute the inverse on the GPU invoking the four routines developed and finally, transfer the inverse to the host.

Hybrid implementation: LU(Hyb).

This implementation is based on a proposal from Barrachina et. al [9]. In that work the authors proposed a hybrid computing strategy (using the GPU and the CPU) to improve the eficiency of routine `getrf`.

The resulting `getrf` implementation uses the CPU to compute the fine-grained operations, like the factorization of the current column panel, while the GPU computes the update of the trailing submatrix.

2.2 Matrix inversion via Gauss-Jordan elimination

The Gauss-Jordan elimination algorithm [4] (GJE) for matrix inversion is, in essence, a reordering of the computation 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 [6, 3, 15]. 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 [6, 3]. 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 the procedure can be cast in terms of the matrix-matrix product, an operation with a high parallelism. Therefore, GJE is a highly appealing method for matrix inversion on emerging architectures like GPUs, where many computational units are available, specially if a highly-tuned implementation of the matrix-matrix product is available (e.g. Volkov's `gemm` [10] included in the CUBLAS library).

Four implementations for the GJE method (with partial pivoting) on two parallel architectures are presented: a multi-core CPU architecture and a GPU from NVIDIA. The following variants differ on which part of the computation is performed on the CPU (the general-purpose processor or host), and which part is off-loaded to the hardware accelerator (the GPU or device). They all try to reduce the number of communications between the memory spaces of the host and the device. The first three versions were presented in Benner et al. [2] work, while the last version is an original contribution.

Implementation on a multi-core CPU: GJE(CPU)**.**

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

```
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 $b \times b$

    $\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} := GJE_{\mathrm{UNB}}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$    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.

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 moved back to the host. Again, all the parallelism is extracted from a multi-threaded implementation of BLAS (e.g. the implementation from NVIDIA, CUBLAS).

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 low performance on massively parallel architectures like GPUs. To solve this problem, Benner et al.

[2] proposed a hybrid version in which operations are performed in the most convenient device, exploiting the capabilities of both architectures.

In this case, the matrix is initially transferred to the device; at the beginning of each iteration of the algorithm in Figure 1, the current column panel, composed by blocks $\left[A_{01}^T, A_{11}^T, A_{21}^T\right]^T$ is moved to the CPU and factorized there; the result is immediately transferred back to the device, where all the remaining computations (matrix-matrix products) are performed. This pattern is repeated until the full matrix inverse is computed. The inverse is finally transferred from the device memory to the host.

In summary, only the factorization of the current column panel 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).

Concurrent Hybrid implementation: GJE(Hyb-Con)**.**

Although the previous version (GJE(Hyb)) achieves an important computational efficiency due to the fact that each operation is executed on the most convenient device, all stages are executed sequentially.

We propose a new implementation of algorithm in Figure 1 in which the execution of the different stages of the method are performed concurrently. Thus, GJE(Hyb-Con) extracts parallelism from the use of multithreaded implementations of BLAS (e.g. MKL, CUBLAS) and from the concurrent execution of operations in both architectures (CPU and GPU).

In this case, the matrix is transferred to the device, and operations of algorithm in Figure 1 are reordered:

1. The current column panel ($[A_{01}^T; A_{11}^T; A_{21}^T]$) is transferred to the host and factorized there.
2. $[A_{01}^T; A_{11}^T; A_{21}^T]$ are transferred to the GPU.
3. $[A_{02}^T; A_{12}^T; A_{22}^T]$ are updated on the GPU.
4. The first b columns of blocks $[A_{02}^T; A_{12}^T; A_{22}^T]$ (that is, blocks $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]$ of the next iteration) are transferred to the host.
5. While the GPU updates blocks $[A_{00}^T; A_{10}^T; A_{20}^T]$, the CPU factorizes $[\hat{A}_{01}^T; \hat{A}_{11}^T; \hat{A}_{21}^T]$.
6. Repeat steps 2–6 until the full matrix inverse is computed.

In summary, this new implementation executes every operation on the most convenient architecture and overlap the update of $[A_{00}^T; A_{10}^T; A_{20}^T]$ on GPU with the factorization of $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$ on CPU.

3 Experimental results

In this section we evaluate the parallel implementations described in Section 2 for the computation of a matrix inverse:

– Implementations based on the LU factorization: LU(CPU), LU(GPU), and LU(Hyb).
– Implementations based on the GJE method: GJE(CPU), GJE(GPU), GJE(Hyb), and GJE(Hyb-Con).

The target platform consists of two Intel Xeon QuadCore processors connected to a Tesla C1060 GPU. Table I offers more details on the hardware. Note that, although the peak performance for a Tesla C1060 is 933 GFLOPS, in practice, only few applications can take profit from all the architecture capabilities. This is not the case of linear algebra operations (like matrix inversion); e.g., a well suited operation for parallel programming like the matrix-matrix product achieves a performance of approximately 350 GFLOPS on a Tesla C1060 [11, 12].

The Intel MKL 10.1 implementation of BLAS [7] and LAPACK is employed to compute most of the operations on the general-purpose processor, while the NVIDIA CUBLAS (version 2.1) is employed on the GPU.

We set OMP_NUM_THREADS to 8 so that one thread is employed per core in the parallel execution of the MKL routines in the two Intel Xeon QuadCore processors.

All experiments employ single precision floating-point arithmetic, and all results include the communication times between the host and the device memory spaces.

Processors	#cores	Frequency (GHz)	L2 cache (MB)	Memory (GB)	Single precision peak performance (GFLOPS)
Intel Xeon QuadCore E5520	8	2.27	8	24	147.15
Nvidia TESLA c1060	240	1.3	–	4	933.0

Table 1. Hardware employed in the experiments.

Experiments employed matrices with dimensions between 1000 and 8000. The different implementations of matrix inversion were evaluated with several block sizes (32, 64, 96, 128, 192, 256, 288, 320, 512, and 1024), but for simplicity, only the results obtained with the optimal block size are showed.

Figure 2 shows the execution time (left) and performance (right), measured in seconds and GFLOPS (10^9 flops per second), respectively, attained by the different implementations for matrix inversion based on the LU factorization.

The LU(CPU) variant achieves the best results for small and medium matrices. This is because no communication time is needed and also because the features of the implemented algorithm do not permit to exploit all the capabilities of a massively parallel architecture like the GPU. LU(Hyb) is the best LU based implementation for large matrices. Since large problems involve a great number of floating point operations and data, they are more suitable for the GPU architecture and, therefore, obtain higher performance.

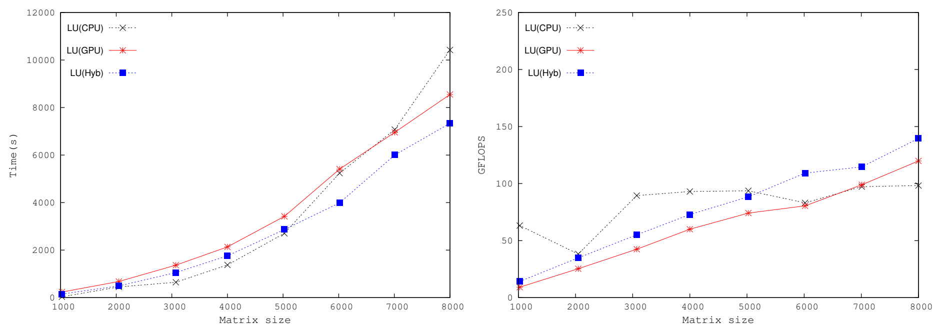

Fig. 2. Execution time (left) and GFLOPS (right) attained by implementations based on Gaussian elimination.

Figure 3 shows the execution time and performance for the implementations based on the GJE technique. Again, the GJE(CPU) implementation achieves the best execution time for small matrices. The reasons exposed in the previous experiments are also valids here. As argued in section 2.2, the GJE algorithm is more suitable for massively parallel architectures than the LU based algorithm. As a consequence, all the GJE implementations for the GPU presented perform better than the best LU based variant. The GJE(Hybrid) approach outperforms GJE(GPU) for small and medium matrices (due to the execution of each operation on the most convenient device), but for larger matrices, the cost of the fine-grain operations becomes less relevant and, therefore, the gain of the Hybrid approach becomes smaller than the extra time needed by data transfers. Finally, we can remark the improvement introduced by GJE(Hyb-Con). This version introduces a second level of parallelism, executing operations concurrently on the CPU and on the GPU. Thus, we exploit all the capabilities of the platform attaining the higher performance.

In summary:

– Codes for GPU are notoriously faster than codes for the CPU.
– Gains from GPU codes are lower for the traditional approach based on the LU factorization. This is because the GJE algorithm is well suited for its implementation in a massively parallel architecture like the GPU.
– The best implementation of each algorithm is an hybrid implementation. This demonstrates the benefit of executing each task on the most convenient device despite of communications overheads.
– GJE(HybCon) is the best implementation and is two times faster than the implementation provided by LAPACK, attaining 220GFLOPS for matrices

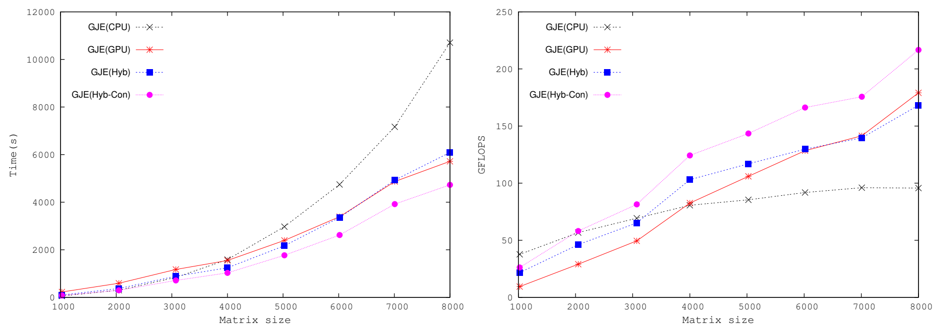

Fig. 3. Execution time (left) and GFLOPS (right) attained by implementations based on Gaussian-Jordan elimination.

with dimension 8000. Note that, even the GPU peak performance is 933 GFLOPS, it is unreachable for most of the applications. A more real reference is the peak rate attained by matrix-matrix product implementations. The implementation from NVIDIA included in the CUBLAS library reaches 350GFLOPS on a Tesla C1060 [11], but we can find slightly better implementations at [10, 12]. A more real comparison can be done with operations like the LU [13] and Cholesky [14] factorizations, which are more similar to the matrix inversion than the matrix-matrix product.

4 Concluding remarks and Future Work

We have demonstrated the benefits of using a GPU to off-load part of the computations in a dense linear algebra operation rich in level-3 BLAS like the matrix inversion.

The evaluation of matrix inversion codes clearly identify the superior performance of the procedures based on Gauss-Jordan elimination over Gaussian elimination (i.e., LU factorization).

The proposed implementation for the algorithm based on GJE, GJE(Hyb-Con), delivers the highest performance and shows a good scalability.

Several questions about the computational performance improvement of matrix inversion with GPUs could be explored in more detail in the future:

– Double precision arithmetic is required in some applications. Performance of current GPUs in double precision arithmetic is considerably lower, but refinement techniques that given a single precision solution obtain a double precision solution can be explored.
– Automatic procedures to obtain the optimal block size for a given matrix.
– Study the possibility of extending the work to a distributed memory scenary.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, third edition, 1999.
2. P. Benner, P. Ezzatti, E. S. Quintana, and A. Remón. Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function. In *Lecture Notes in Computer Science, 7th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks – HeteroPar'09*, 2009.
3. P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.
4. A. V. Gerbessiotis. Algorithmic and practical considerations for dense matrix computations on the BSP model. PRG-TR 32, Oxford University Computing Laboratory, 1997.
5. G.H. Golub and C.F. Van Loan. *Matrix computations.* The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.
6. J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.
7. Intel Corporation., `http://www.intel.com/`.
8. E.S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R.A. van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22:1762–1771, 2001.
9. F. Igual R. Mayo E. S. Quintana G. Quintana S. Barrachina, M. Castillo. Exploting the capabilities of modern gpus for dense matrix computations. *Concurrency and Computation: Practice & Experience*, 21:2457–2477, 2009.
10. V. Volkov, J. Demmel. LU, QR, and Cholesky factorizations using vector capabilities of GPUs. *LAPACK Working Note 202.*
11. Lung-Sheng Chien. Hand-Tuned SGEMM in GT200 GPU. *Tech. Report (Tsing Hua University).*
12. R. Nath, S. Tomov, J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings Vecpar'10*
13. S. Tomov, J. Dongarra M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing (to appear).*
14. G. Quintana-Ortí F.D. Igual E. S. QuintanaV. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *FLAME Working Note 32.*
15. The University of Texas at Austin, `http://www.cs.utexas.edu/~flame/`.