Using Hybrid CPU-GPU Platforms to Accelerate the Computation of the Matrix Sign Function

Peter Benner[1], Pablo Ezzatti[2],
Enrique S. Quintana-Ortí[3], and Alfredo Remón[3]

[1] Fakultät für Mathematik, Chemnitz University of Technology,
D-09107 Chemnitz, Germany
benner@mathematik.tu-chemnitz.de
[2] Centro de Cálculo–Instituto de la Computación, Universidad de la República,
11.300–Montevideo, Uruguay
pezzatti@fing.edu.uy
[3] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I,
12.071–Castellón, Spain
{quintana,remon}@icc.uji.es

Abstract. We investigate the numerical computation of the matrix sign function of large-scale dense matrices. This is a common task in various application areas. The main computational work in Newton's iteration for the matrix sign function consits of matrix inversion. Therefore, we investigate the performance of two approaches for matrix inversion based on Gaussian (LU factorization) and Gauss-Jordan eliminations. The target architecture is a current general-purpose multi-core processor connected to a graphics processor. Parallelism is extracted in both processors by linking sequential versions of the codes with multi-threaded implementations of BLAS. Our results on a system with two Intel Quad-Core processors and an NVIDIA Tesla C1060 illustrate the performance and scalability attained by the codes on this system.

Keywords: Matrix sign function, hybrid platforms, GPUs, multi-core processors, linear algebra, high performance computing.

1 Introduction

Consider a matrix $A \in \mathbb{R}^{n \times n}$ with no eigenvalues on the imaginary axis, and let

$$A = T^{-1} \begin{pmatrix} J_- & 0 \\ 0 & J_+ \end{pmatrix} T, \tag{1}$$

be its Jordan decomposition, where the eigenvalues of $J_- \in \mathbb{R}^{j \times j}$ /$J_+ \in \mathbb{R}^{(n-j)\times(n-j)}$ all have negative/positive real parts [1]. The *matrix sign function* of A is then defined as

$$\operatorname{sign}(A) = T^{-1} \begin{pmatrix} -I_j & 0 \\ 0 & I_{n-j} \end{pmatrix} T, \tag{2}$$

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 132–139, 2010.

where I denotes the identity matrix of the order indicated by the subscript. The matrix sign function is a useful numerical tool for the solution of control theory problems (model reduction, optimal control) [2], and the bottleneck computation in many lattice quantum chromodynamics computations [3] and dense linear algebra computations (block diagonalization, eigenspectrum separation) [1,4]. Large-scale problems as those arising, e.g., in control theory often involve matrices of dimension $n \rightarrow O(10,000 - 100,000)$ [5].

There are simple iterative schemes for the computation of the sign function. Among these, the Newton iteration, given by

$$\begin{aligned} A_0 &:= A, \\ A_{k+1} &:= \tfrac{1}{2}(A_k + A_k^{-1}),\ k = 0, 1, 2, \ldots, \end{aligned} \tag{3}$$

is specially appealing for its simplicity, efficiency, parallel performance, and asymptotic quadratic convergence [4,6]. However, even if A is sparse, $\{A_k\}_{k=1,2,\ldots}$ in general are full dense matrices and, thus, the scheme in (3) roughly requires $2n^3$ floating-point arithmetic operations (flops) per iteration.

In the past, large-scale problems have been tackled using message-passing parallel solvers based on the matrix sign function which were then executed on clusters with a moderate number of nodes/processors [7]. The result of this effort was our message-passing library `PLiC` [8] and subsequent libraries for model reduction (`PLiCMR`, see [9]) and optimal control (`PLiCOC`, see [10]). Using this library, 16–32 processors showed to provide enough computational power to solve problems with $n \approx 10,000$ in a few hours.

Following the recent uprise of hardware accelerators, like the graphics processors (GPUs), and the increase in the number of cores of current general-purpose processors, in this paper we evaluate an alternative approach that employs a sequential version of the codes in the `PLiC` library, and extracts all parallelism from tuned multi-threaded implementations of the BLAS (*Basic Linear Algebra Subprograms*) [11,12,13]. The results attained in a hybrid, heterogeneous architecture composed of a general-purpose multi-core processor and a GPU demonstrate that this is a valid platform to deal with large-scale problems which, only a few years ago, would have required a distributed-memory cluster.

The rest of the paper is structured as follows. In Section 2 we elaborate on the hybrid computation of the matrix inverse on a CPU-GPU platform. This is followed by experimental results in Section 3, while concluding remarks and open questions follow in Section 4.

2 High-Performance Matrix Inversion

As equation (3) reveals, the application of Newton's method to the sign function requires, at each iteration, the computation of a matrix inverse. We next review two different methods for the computation of this operation, based on the LU factorization and Gauss-Jordan transformations.

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 Gaussian elimination (i.e., the LU factorization), and consist of the following three steps:

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

LAPACK [14] is a high-performance linear algebra library which provides routines that cover the functionality required in the previous steps. In particular, routine `getrf` yields 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 inverse following the previous four steps is $2n^3$ flops. The algorithm sweeps through the matrix four times (one per step) and presents a mild load imbalance, due to the work with the triangular factors.

2.2 Matrix Inversion via Gauss-Jordan Elimination

The Gauss-Jordan elimination algorithm [15] (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 [16,17,18]. 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 [16,17]. (A description of the unblocked version, called from inside the blocked one, can be found in [19]; 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, provided a highly-tuned implementation of the matrix-matrix product is available.

We next introduce three implementations for the GJE method (with partial pivoting) on two parallel architectures: a multi-core CPU architecture and a GPU from NVIDIA. The following variants differ on which part of the computations 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.

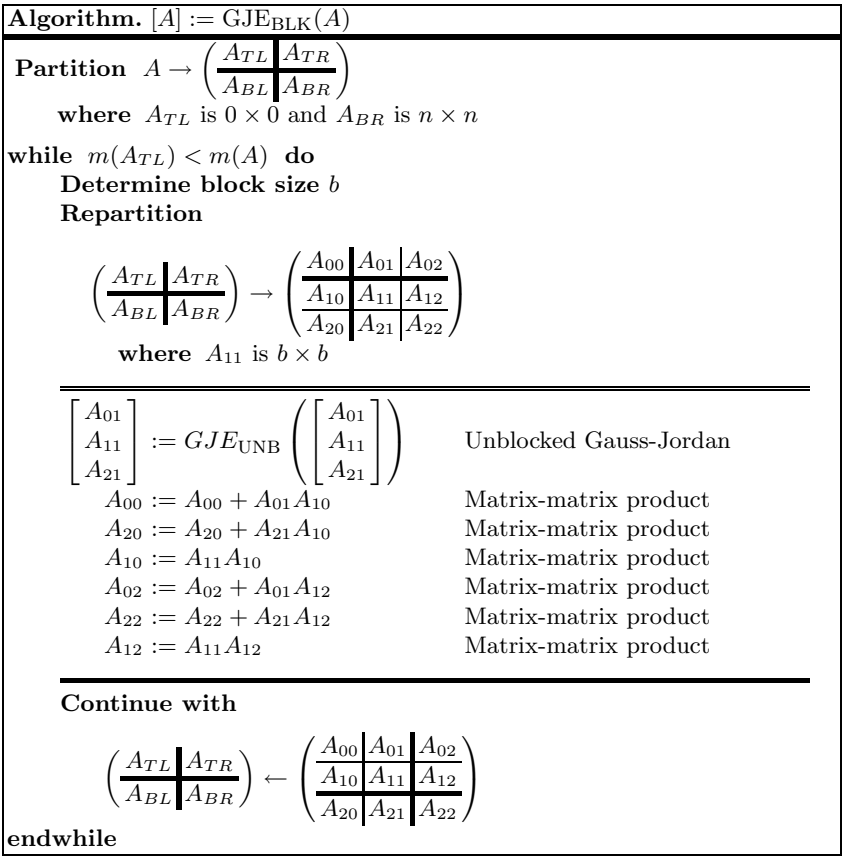

Fig. 1. Blocked algorithm for matrix inversion via GJE without pivoting

Implementation on a multi-core CPU: GJE(CPU)**.** In this first variant all operations are performed on the CPU. Parallelism is obtained from a multi-threaded implementation of BLAS for general-purpose processors. Since most of the computations are cast in terms of products of matrices, high performance can be expected from this variant.

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 proceed there next; and the result (the matrix inverse) is finally moved back to the host.

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, as the low computational cost and data dependencies deliver low performance on massively parallel architectures like the GPU. To solve this problem, we propose a hybrid implementation. In this new

approach, operations are performed in the most convenient device, exploiting the capabilities of both architectures.

In particular, in this variant 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 of $\left[A_{01}^T, A_{11}^T, A_{21}^T\right]^T$ is moved to the CPU and factorized there. The result is immediately transferred back to the device, where all 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 BLAS-1 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.

3 Experimental Results

In this section we evaluate four parallel multi-threaded codes to compute the inverse of a matrix:

- LAPACK(CPU): The four steps of the LAPACK approach, with all computations carried out on the CPU and parallelism extracted by using a multi-threaded implementation of BLAS; see subsection 2.1.
- GJE(CPU), GJE(GPU), and GJE(Hybrid): The implementations described in subsection 2.2.

Two different implementations of the BLAS (Goto BLAS [20] –version 1.26– and Intel MKL [21] –version 10.1–) were used to execute operations on the general-purpose processor, while on the NVIDIA GPU, CUBLAS [22] (version 2.1) was the library that we used.

The experiments employ single and double precision and the results always include the cost of data transfers between the host and device memory spaces. The target platform consists of two Intel Xeon QuadCore processors connected to an NVIDIA Tesla C1060. Table 1 offers more details on the hardware.

Figure 2 reports the GFLOPS (10^9 flops per second) rates attained by the different implementations of the inversion codes operating on single-precision matrices with sizes between 1,000 and 8,000. Several algorithmic block sizes

Table 1. Hardware employed in the experiments

Processors	#cores	Frequency (GHz)	L2 cache (MB)	Memory (GB)	Single/Double precision peak performance (GFLOPS)
Intel Xeon QuadCore E5405	8	2.3	12	8	149.1/74.6
Nvidia TESLA c1060	240	1.3	–	4	933.0/78.0

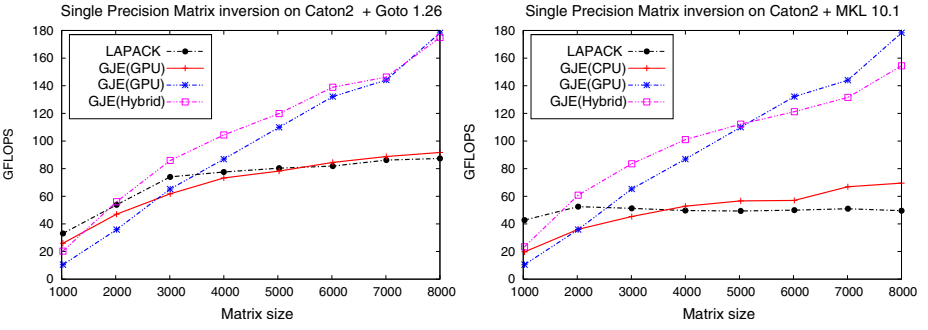

Fig. 2. Performance of the matrix inversion codes

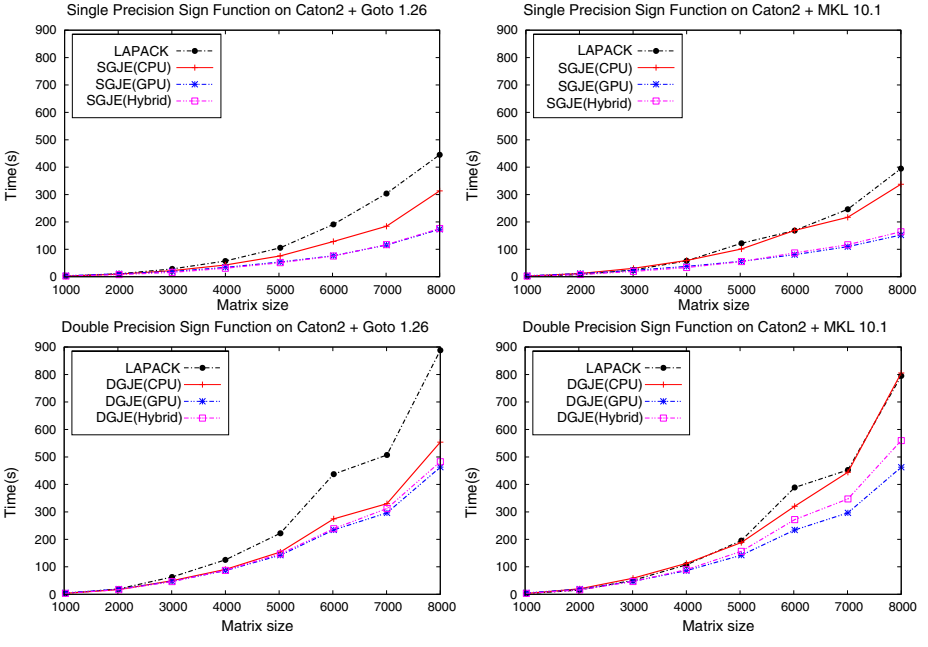

Fig. 3. Execution times of the Newton iteration for the matrix sign function with the matrix inversion implemented using the different variants discussed

(parameter b in Figure 1) were tested but, for simplicity, the results in all figures, hereafter, correspond to those obtained with the optimal block size.

The LAPACK code executed using all 8 cores of the two general-purpose processors yields the lowest GFLOPS rate, while the GJE algorithm using the same resources performs slightly better. Both implementations that employ the GPU outperform the ones executed only on the CPU. The Hybrid approach is the best option for small/medium matrices, while the version executed entirely on the GPU is the best for large matrices.

Figure 3 shows execution times of the Newton's iteration for the matrix sign function, using the previous matrix inversion codes and both single and double precision data. As expected, the LAPACK implementation delivers the highest execution time, followed by GJE(CPU). Codes for GPU are notoriously/slightly faster in single/double precision. Gains from GPU codes are larger for single precision computations and for large matrices.

4 Concluding Remarks and Future Work

We have demonstrated the benefits of using a current GPU to off-load part of the computations in a dense linear algebra operation rich in level-3 BLAS like the matrix inversion. This operation is the basis for the computation of the matrix sign function via Newton's iteration and is also the key to the efficient solution of important problems in control theory such as model reduction and optimal control.

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

Our research poses some open questions which form the basis of our ongoing and future work:

- Most applications in control theory and linear algebra require double precision but current GPUs deliver considerable lower performance when they operate with this data type. Is it possible to compute the sign function in single precision and then use iterative refinement [23] to obtain a double precision solution at a low cost?
- Can we overlap CPU and GPU computations so that while the CPU is computing some blocks the GPU updates others? Note that this requires a careful synchronization of the data transfers between the memory spaces of host and device.
- Is it possible to overlap computation on the GPU with data transfers between the CPU and the GPU memory spaces to improve the performance for the small problem sizes?

Acknowledgments

This work was supported by the Spanish Ministry of Science and Innovation/ FEDER (contract TIN2008-06570-C04-01), by the Fundación Caixa-Castelló/ Bancaixa (contracts no. P1B-2007-19 and P1B-2007-32) and by the Generalitat Valenciana (contract PROMETEO/2009/013).

References

1. Golub, G., Loan, C.V.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore (1996)
2. Petkov, P., Christov, N., Konstantinov, M.: Computational Methods for Linear Control Systems. Prentice Hall, Hertfordshire (1991)

3. Frommer, A., Lippert, T., Medeke, B., Schilling, K. (eds.): Numerical Challenges in Lattice Quantum Chromodynamics. Lecture Notes in Computational Science and Engineering, vol. 15. Springer, Heidelberg (2000)
4. Byers, R.: Solving the algebraic Riccati equation with the matrix sign function. Linear Algebra Appl. 85, 267–279 (1987)
5. IMTEK, Oberwolfach model reduction benchmark collection, http://www.imtek.de/simulation/benchmark/
6. Benner, P., Quintana-Ortí, E.: Solving stable generalized Lyapunov equations with the matrix sign function. Numer. Algorithms 20, 75–100 (1999)
7. Benner, P., Claver, J., Quintana-Ortí, E.: Parallel distributed solvers for large stable generalized Lyapunov equations. Parallel Processing Letters 9, 147–158 (1999)
8. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: A portable subroutine library for solving linear control problems on distributed memory computers. In: Cooperman, G., Jessen, E., Michler, G. (eds.) Workshop on Wide Area Networks and High Performance Computing, Essen (Germany), September 1998. Lecture Notes in Control and Information, pp. 61–88. Springer, Heidelberg (1999)
9. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: State-space truncation methods for parallel model reduction of large-scale systems. Parallel Comput. 29, 1701–1722 (2003)
10. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: Solving linear-quadratic optimal control problems on parallel computers. Optimization Methods & Software 23, 879–909 (2008)
11. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Soft. 5, 308–323 (1979)
12. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN basic linear algebra subprograms. ACM Trans. Math. Soft. 14, 1–17 (1988)
13. Dongarra, J.J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. 16, 1–17 (1990)
14. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
15. Gerbessiotis, A.V.: Algorithmic and Practical Considerations for Dense Matrix Computations on the BSP Model. PRG-TR 32, Oxford University Computing Laboratory (1997)
16. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal linear algebra methods environment. ACM Trans. Math. Soft. 27, 422–455 (2001)
17. Bientinesi, P., Gunnels, J.A., Myers, M.E., Quintana-Ortí, E.S., van de Geijn, R.A.: The science of deriving dense linear algebra algorithms. ACM Trans. Math. Soft. 31, 1–26 (2005)
18. University of Texas, http://www.cs.utexas.edu/~flame/
19. Quintana-Ortí, E., Quintana-Ortí, G., Sun, X., van de Geijn, R.: A note on parallel matrix inversion. SIAM J. Sci. Comput. 22, 1762–1771 (2001)
20. Texas Advanced Computing Center, http://www.tacc.utexas.edu/~kgoto/
21. Intel Corporation, http://www.intel.com/
22. Nvidia Corporation, http://www.nvidia.com/cuda/
23. Nicholas, J.N.: Accuracy and stability of numerical algorithms, Philadelphia, PA, USA (1996)