

All



ADVANCED SEARCH

Conferences > 2011 19th International Eurom... ?

# High Performance Matrix Inversion on a Multi-core Platform with Several GPUs

Publisher: IEEE [Cite This](#) PDF

Pablo Ezzatti ; Enrique S. Quintana-Ortí ; Alfredo Remon **All Authors** ...

**11**  
Cites in  
Papers

**508**  
Full  
Text Views

## Abstract

### Document Sections

- I. Introduction
- II. Gauss-jordan Elimination Algorithm For Matrix Inversion
- III. Matrix Inversion On a Hybrid Multi-gpu Platform
- IV. Experimental Results
- V. Related Work

Show Full Outline ▾

Authors

Figures

References

Citations

Keywords

Metrics

More Like This

## Abstract:

Inversion of large-scale matrices appears in a few scientific applications like model reduction or optimal control. Matrix inversion requires an important computational e... **View more**

## ▼ Metadata

### Abstract:

Inversion of large-scale matrices appears in a few scientific applications like model reduction or optimal control. Matrix inversion requires an important computational effort and, therefore, the application of high performance computing techniques and architectures for matrices with dimension in the order of thousands. Following the recent uprise of graphics processors (GPUs), we present and evaluate high performance codes for matrix inversion, based on Gauss-Jordan elimination with partial pivoting, which off-load the main computational kernels to one or more GPUs while performing fine-grain operations on the general-purpose processor. The target architecture consists of a multi-core processor connected to several GPUs. Parallelism is extracted from parallel implementations of BLAS and from the concurrent execution of operations in the available computational units. Numerical experiments on a system with two Intel QuadCore processors and four NVIDIA c1060 GPUs illustrate the efficiency and the scalability of the different implementations, which deliver over  $1.2 \times 10^{12}$  floating point operations per second.

**Published in:** 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing

**Date of Conference:** 09-11 February 2011

**DOI:** 10.1109/PDP.2011.66

**Date Added to IEEE Xplore:** 24 March 2011

**Publisher:** IEEE

**Print ISBN:**978-1-4244-9682-2

**Conference Location:** Ayia Napa, Cyprus

## ▼ ISSN Information:

Citations are not available for this document.

Contents

























## SECTION I.

### Introduction

Inversion of large-scale matrices is required in a few scientific applications (model reduction, polar decomposition, optimal control and prediction, statistics, ...). As numerically stable procedures for its computation exhibit a computational cost of  $O(n^3)$  flops (floating-point arithmetic operations), this calls for the application of high performance computing in the inversion of matrices of dimension in the order of thousands.

In the recent years, different works have demonstrated the potential of GPUs (graphics processors) to yield high performance on dense linear algebra operations that can be cast in terms of matrix-matrix products [1], [2]. This is the case of matrix inversion via the Gauss-Jordan elimination algorithm, which was implemented on a platform with a single GPU in [3]. In this paper we extend previous work, presenting several algorithms for the computation of a matrix inverse on a hybrid platform consisting of one (or more) multi-core processor(s) connected to several GPUs. The implementations exploit the capabilities of the hybrid resources in the platform, attaining remarkable high performance.

The paper is structured as follows. In Section II we present the Gauss-Jordan elimination algorithm. This is followed by the description of several implementations of this algorithm on the target platform in Section III, and numerical results in Section IV. Section V briefly introduces related works and its applicability to the matrix inversion problem. Concluding remarks and open questions close the paper in Section VI.

In the paper, we will use the Matlab semicolon notation to refer to blocked matrices; thus, e.g.,  $[U; V]$  will refer to  $[U^T, V^T]^T$ .

## SECTION II.

### Gauss-jordan Elimination Algorithm For Matrix Inversion

The Gauss-Jordan elimination algorithm (GJE) for matrix inversion is, in essence, a reordering of the computations performed by the traditional matrix inversion method based on Gaussian elimination; hence both procedures feature the same arithmetic cost:  $2n^3$  flops [4], [5]. Figure 1 illustrates a blocked version of the GJE algorithm for matrix inversion using the FLAME notation. There  $m(A)$  stands for the number of rows of a matrix  $A$ . We believe the rest of the notation to be intuitive; for further details, see [6], [7]. A description of the unblocked version of the algorithm, GJE\_UNB, called from inside the blocked one, can be found in [5]; for simplicity, in Figure 1, we do not include the application of pivoting during the factorization, but details can be found there as well.



















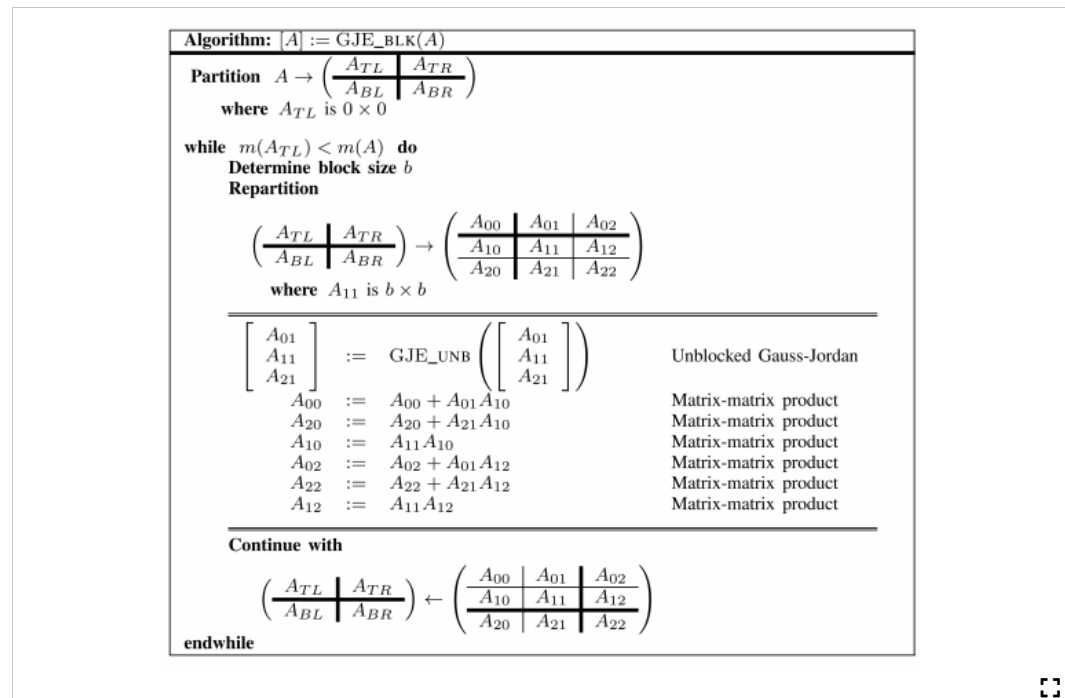
The traditional method for the computation of a matrix inverse is based on Gaussian elimination and presents some inappropriate features for high performance parallel computing. In particular, this approach operates with triangular matrices which often yields load imbalance. On the other hand, the major advantages of GJE over the traditional approach for matrix inversion via Gaussian elimination are:

- Most of the computations in GJE can be cast in terms of matrix-matrix products. The product of matrices is a highly parallel operation that can exploit all the capabilities of massively parallel architectures like GPUs.
- The updates of the blocks of the matrix in positions other than the panel factored during the current iteration can be all performed in parallel.

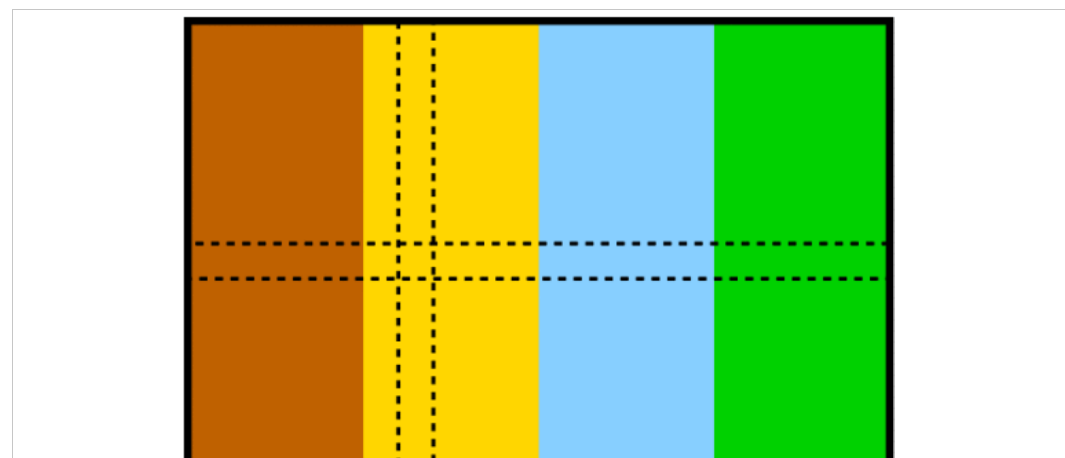
### SECTION III.

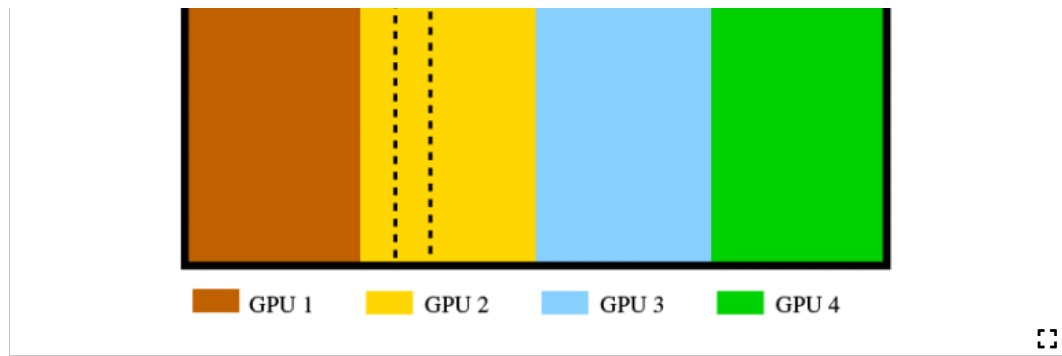
## Matrix Inversion On a Hybrid Multi-gpu Platform

As argued, GJE is a highly-appealing parallel method for matrix inversion and, therefore, suitable for emerging architectures like GPUs, where many computational units are available.



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





**Fig. 2.** Data distribution for the  $GJE_{mGPU}$  implementation on 4 GPUs.

The hybrid target platform considered in this work consist of two different architectures: one (or more) general-purpose multi-core processors and a few GPUs. Although most of the operations performed by algorithm  $GJE\_BLK$  are suitable for the GPU architecture, a few can be more efficiently executed on a general-purpose processor. This is the case of the panel factorization of  $[A_{01}; A_{11}; A_{21}]$ , computed via the Gauss-Jordan unblocked algorithm. This operation exhibits a low computational cost and presents many data dependencies (specially, when pivoting for numerical stability is introduced), which often yields low performance on massively parallel architectures like GPUs, but an acceptable performance on a general-purpose multi-core processor.

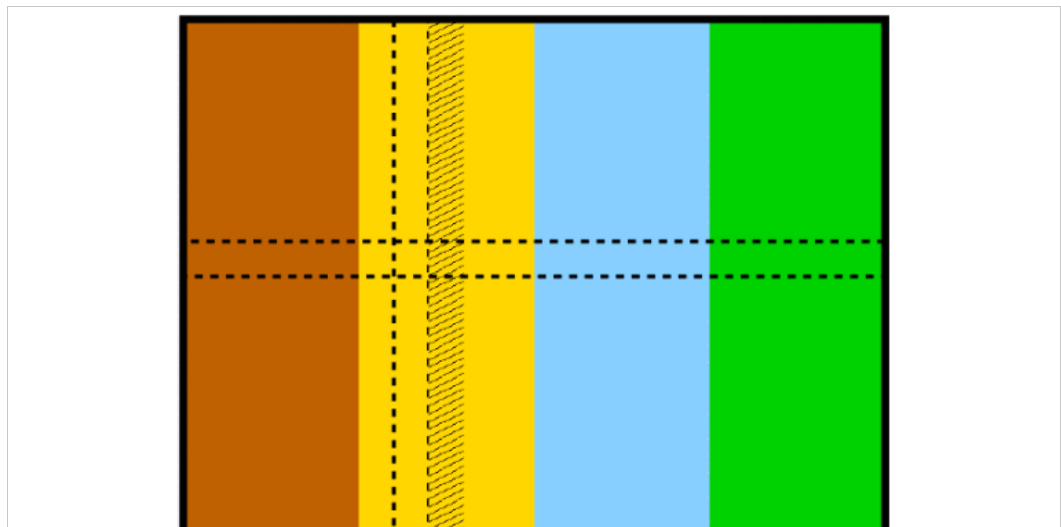
The properties of the GJE algorithm (Figure 1) and the hybrid nature of the platform allow us to extract parallelism at two levels: at the bottom level, each matrix-matrix product in the main loop of algorithm  $GJE\_BLK$  can be executed in parallel, using a multi-threaded implementation of this operation (available, e.g., in most BLAS implementations for general-purpose multi-core processors and many-core GPUs). On top of this, once the current panel consisting of  $[A_{01}; A_{11}; A_{21}]$  is factored, all updates corresponding to the matrix-matrix products can be computed concurrently.

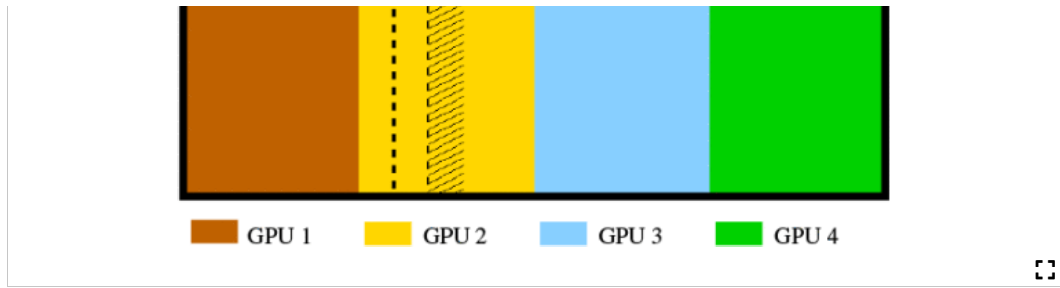
In the following, we describe five different implementations of GJE-based matrix inversion codes, which exploit the hybrid architecture of the target platform. A goal for all these variants is that the communication overhead introduced by data transfers through the PCI-e bus that connects CPU and GPU is maintained as low as possible.

### 1) Hybrid implementation ( $GJE_{mGPU}$ )

In previous work we proposed an implementation for the GJE algorithm on a hybrid platform consisting of a multi-core processor and a single GPU [3]. There, we demonstrated the benefits of executing each operation on the most convenient device, exploiting the capabilities of both architectures (CPU and GPU).

$GJE_{mGPU}$  is the naive translation of this single GPU implementation to the multi-GPU case. In  $GJE_{mGPU}$ , the matrix-matrix products in algorithm  $GJE\_BLK$  are equally distributed among all the available GPUs, while the fine grain factorization of the panel is executed on the CPU.





**Fig. 3.** Data distribution between GPUs for the  $GJE_{LA}$  implementation.

In particular, consider an initial block column partitioning of the matrix  $A$  into  $k = 4$  blocks, where  $k$  equals the number of available GPUs (See Figure 2). Then, the algorithm proceeds as follows:

- a)  $\forall i \leq k$ , the  $i$ th block is transferred to the  $i$ th GPU.
- b) The current column panel  $[A_{01}; A_{11}; A_{21}]$ , consisting of  $b$  columns, is transferred to the CPU and factored there. The resulting factor is broadcasted to all GPUs.
- c)  $[A_{02}; A_{12}; A_{22}]$  are updated in collaboration by the GPUs. Each GPU is responsible for the updates to its local block. Note that no communication is required in this step.
- d)  $[A_{00}; A_{10}; A_{20}]$  are updated in collaboration by the GPUs. The comments from step c) also apply here.
- e) Move the factorization forward by  $b$  columns and repeat steps b)-e) until the full matrix inverse is computed.
- f) All the GPUs transfer their corresponding column block to the host.

In summary, this implementation executes each operation on the most convenient architecture. The factorization of the current panel is executed on the CPU, since it involves a reduced number of data (limited by the algorithmic block size), pivoting and level 1/2 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 GPUs. Also, this enables overlapping the update of  $[A_{00}; A_{10}; A_{20}]$  on the GPUs with the factorization of  $[A_{01}; A_{11}; A_{21}]$  corresponding to the next iteration on the CPU.

## 2) Look-Ahead implementation ( $GJE_{LA}$ )

The  $GJE_{mGPU}$  variant limits the amount of overlapped computations in both architectures, specially in the initial iterations where the updates of blocks to the left of the current column panel require few operations. The  $GJE_{LA}$  implementation reduces the computational time increasing the concurrent execution of operations on the CPU and the GPUs as described next.

In this variant we apply some minor changes to obtain a look-ahead variant [8]:

- a)  $\forall i \leq k$ , the  $i$ th column panel is transferred to the  $i$ th GPU.
- b) The first column panel  $[A_{01}; A_{11}; A_{21}]$  is transferred to the CPU and factored there. The resulting factor is broadcasted to all GPUs.
- c) The first  $b$  columns of block  $[A_{02}; A_{12}; A_{22}]$  (that is, block  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$  of the next iteration) are updated and transferred to the CPU.
- d) While the GPUs update blocks  $[A_{00}; A_{10}; A_{20}]$ , and the remaining part of  $[A_{02}; A_{12}; A_{22}]$ , the CPU factorizes  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$ .



- e) Move the factorization forward by  $b$  columns and repeat steps c)-e) until the full matrix inverse is computed.
- f) All the GPUs transfer their corresponding column block to the host.

Figure 3 shows the data distribution when 4 GPUs are available. The dashed lines mark the current panel while the shaded panel marks block  $[A_{01}; A_{11}; A_{21}]$  referred in steps c)-d).

### 3) Multilevel implementation: (GJE<sub>ML</sub>)

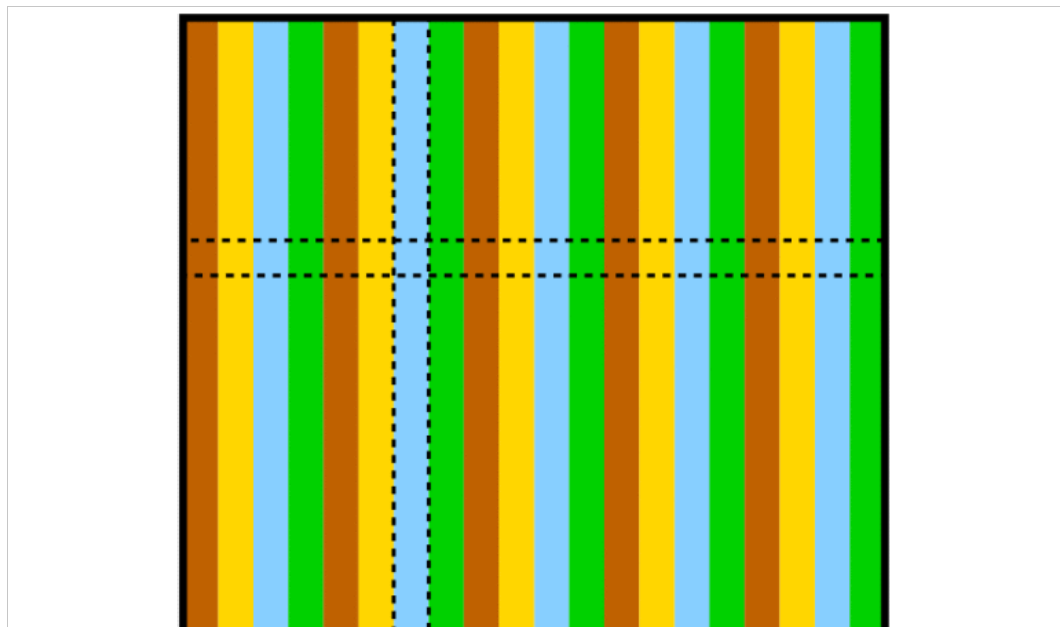
The efficiency of the previous variant is limited by the algorithmic block size. The optimal block size for the products performed on GPUs is usually larger than that for fine/medium-grain operations performed on the CPU. To solve this, the GJE<sub>ML</sub> implementation operates with two block sizes (one for the CPU and one for the GPUs). The goal is to employ the optimal block size independently for each architecture, instead of a unified one.

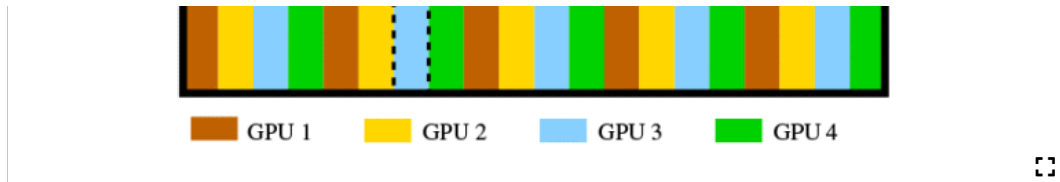
Specifically, this implementation differs from the previous one in the way the panel  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$  is factored. While in previous implementations this block is factored using an unblocked version of the GJE algorithm, in GJE<sub>ML</sub> the CPU executes a blocked version of the algorithm (i.e. GJE\_BLK) to factorize it. Thus, the algorithmic block size employed by the CPU ( $b_c$ ) for this factorization is different from the one employed by GPUs ( $b_g$ ) on their operations, permitting GJE<sub>ML</sub> to adapt its execution to the features of both architectures, by choosing their respective optimal block-sizes  $b_c$  and  $b_g$ .

### 4) Cyclic distribution implementation (GJE<sub>CD</sub>)

The GJE<sub>ML</sub> implementation still presents some problems: the most important is that, at every iteration, the operations performed by one of the GPUs require more time than those carried out by the rest, leading to load imbalance. In particular, the GPU that stores  $[A_{01}; A_{11}; A_{21}]$  is usually the one that also updates (this requires 3 matrix-matrix products) and transfers (to the CPU)  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$  (see Figure 3); this GPU also has to perform 6 matrix-matrix products for the updates of its correspondings parts of blocks  $A_{00}, A_{10}, A_{20}, A_{02}, A_{12}$  and  $A_{22}$ . In summary, this particular GPU computes 9 matrix-matrix products, while the rest of GPUs perform only 3 matrix-matrix products.

Thus, although all GPUs have to perform nearly the same number of arithmetic operations in the algorithm, for the GPU that stores the current column block at each iteration, the work is heavily partitioned and this can significantly decrease the efficiency of the algorithm. Detailed experiments on the inversion of matrices of dimension 20,000 with 4 GPUs, revealed that this GPU requires approximately a 20% more time than the rest of GPUs. This situation has a negative impact on the efficiency because, usually, this GPU stores and updates block  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$ , and it is a critical operation required by the rest of devices to continue with their computations.





**Fig. 4.** Data distribution between GPUs for the  $GJE_{CD}$  implementation.

We can partially overcome this problem by employing a cyclic distribution of  $A$ . The matrix is partitioned in blocks of  $b$  columns, and the  $i$ th block of columns is stored and updated by the  $(i \bmod k)$ ,  $(i \bmod k)$  -th GPU; see Figure 4.

In this version, at an initial stage, the  $i$ th block of columns,  $\forall i \leq k$ , is transferred to the  $(i \bmod k)$  GPU. Then, the matrix inverse is computed as described for  $GJE_{ML}$  and, finally, each GPU transfers its column blocks to the CPU.

Due to the new data distribution, the GPU that requires more time at iteration  $i$  and its analog for iteration  $i + 1$  are different, so we can partially overlap the computations performed at iteration  $i$  by the  $(i \bmod k)$ -th GPU, with computations carried out by the rest of GPUs at iteration  $i + 1$ .

### 5) Merge implementation ( $GJE_{Merge}$ )

Variant  $GJE_{CD}$  increments the GPUs utilization improving the concurrent execution of operations on the GPUs. However, the work performed by each GPU is still unnecessarily partitioned. The performance attained by a GPU when computing a matrix-matrix product depends strongly on the size of the matrices involved, with higher performance being often obtained for matrices of larger dimension. In variant  $GJE_{CD}$ , each GPU performs, at least, six matrix-matrix products and the rows swap required by the pivoting strategy. Additionally, one of the GPUs performs three more matrix-matrix products for the update of  $[\hat{A}_{01}; \hat{A}_{11}; \hat{A}_{21}]$ .

The objective of implementation  $GJE_{Merge}$  is to reduce the number of matrix-matrix products executed by all the GPUs. We can achieve this with a minor change at our algorithm. At every iteration, blocks  $A_{10}$ ,  $A_{12}$  are copied to workspaces  $W_1$  and  $W_2$ , respectively in all GPUs; then the contents of  $A_{10}$ ,  $A_{12}$  are set to zero; finally, a couple of matrix-matrix products are only needed to perform the update of the six blocks, as shown in Figure 5.

In addition, an important improvement in performance can be obtained by merging the copies of blocks  $A_{10}$  and  $A_{12}$  with the swap stage required by pivoting. Thus,  $W_1$  and  $W_2$  will contain blocks  $A_{10}$  and  $A_{12}$  after the pivoting has been applied. This considerably reduces the number of memory accesses and partially hides the overhead introduced by the copies of  $A_{10}$  and  $A_{12}$ .

## SECTION IV.

## Experimental Results

In this section we evaluate the computational performance of matrix inversion with partial pivoting on a platform consisting of two Intel Xeon QuadCore E5410 processors at 2.33GHz, connected to four NVIDIA Tesla C1060 via two PCI-e buses (more details of the hardware employed are given in Table II). All experiments employ single-precision arithmetic. Intel MKL 11.1 for the multi-core CPU and CUBLAS 3.0 for the GPUs are employed.

The implementations described previously are tested and compared with the traditional multi-core implementation via the LAPACK library. This reference implementation computes a matrix inverse in four steps:

- 1) Compute the LU factorization with partial pivoting (routine GETRF).

- 2) Invert a triangular matrix.
- 3) Solve a system with many right-hand sides.
- 4) Undo pivoting (routine GETRI).

In particular, routine GETRF yields the LU factorization (with partial pivoting) of a nonsingular matrix (Step 1), while routine GETRI computes the inverse of a matrix using its LU factorization obtained by GETRF (Steps 2–4).

Also the hybrid implementation proposed at a previous work [3], which only employs a single GPU and a multi-core processor, is included in the study as a reference implementation.

In all these codes, parallelism is extracted from multithreaded implementations of BLAS.

Figure 6 shows the performance in GFLOPS ( $10^9$  flops per second) attained by the five implementations proposed in our work using the 8 cores from the multi-core CPU and 2 of the 4 GPUs available in the platform. For reference we also include the implementation from LAPACK using the 8 cores only.

The new codes outperform the LAPACK implementation even for relatively small matrices. Variant  $GJE_{mGPU}$  performs relatively well, being up to 3.5 times faster than LAPACK, but is still far from the peak performance of the platform. The look-ahead variant achieves an important improvement, being a 15% faster than  $GJE_{mGPU}$ . The use of two block sizes, one per each architecture, introduced at  $GJE_{ML}$  reports moderate benefits. Finally, the best results are obtained by  $GJE_{CD}$  and  $GJE_{Merge}$ ; both implementations are equally efficient in this case. This, indicates that the modifications introduced by  $GJE_{Merge}$  do not provide any gain on 2 GPUs.  $GJE_{CD}$  and  $GJE_{Merge}$  are up to 5.5 times faster than the LAPACK reference implementation achieving up to 550 GFlops.

Figure 7 shows the performance attained by all the variants using all the resources available in the platform (8 cores and 4 GPUs). In this case, variants  $GJE_{mGPU}$  and  $GJE_{LA}$  obtain similar results. The gain attained by  $GJE_{LA}$  is reduced when the number of GPUs is increased, This is because the GPU computational time is also decreased and, consequently, the CPU becomes a bottleneck. In contrast, the modifications introduced by  $GJE_{ML}$  result in a remarkable increment of the performance.  $GJE_{CD}$  reduces the execution time, achieving the 1000 GFlops for the largest matrices. Overall,  $GJE_{Merge}$  is nearly two times faster than the initial implementation,  $GJE_{mGPU}$ . Although it did not yield any gain on 2 GPUs, when 4 GPUs are employed this variant yields a moderate reduction of the computational time. This is due to the fact that the dimension of the operations performed by the GPUs becomes smaller, and thus, merging them results in a higher performance.

**Algorithm:**  $[A] := GJE\_MERGE(A)$

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

**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  $b \times b$

---


$$\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} := GJE\_UNB \left( \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} \right) \quad \text{Unblocked Gauss-Jordan}$$

$$W_1 := A_{10} \quad \text{Copy}$$

$$A_{10} := 0$$

$$\begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} := \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} + \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} W_1 \quad \text{Matrix-matrix product}$$

$$W_2 := A_{12} \quad \text{Copy}$$

$$A_{12} := 0$$

$$\begin{bmatrix} A_{02} \\ A_{12} \\ A_{22} \end{bmatrix} := \begin{bmatrix} A_{02} \\ A_{12} \\ A_{22} \end{bmatrix} + \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} W_2 \quad \text{Matrix-matrix product}$$


---

**Continue with**

$$\left( \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right)$$



Fig. 5. Merge variant of the blocked algorithm for matrix inversion via GJE without pivoting.

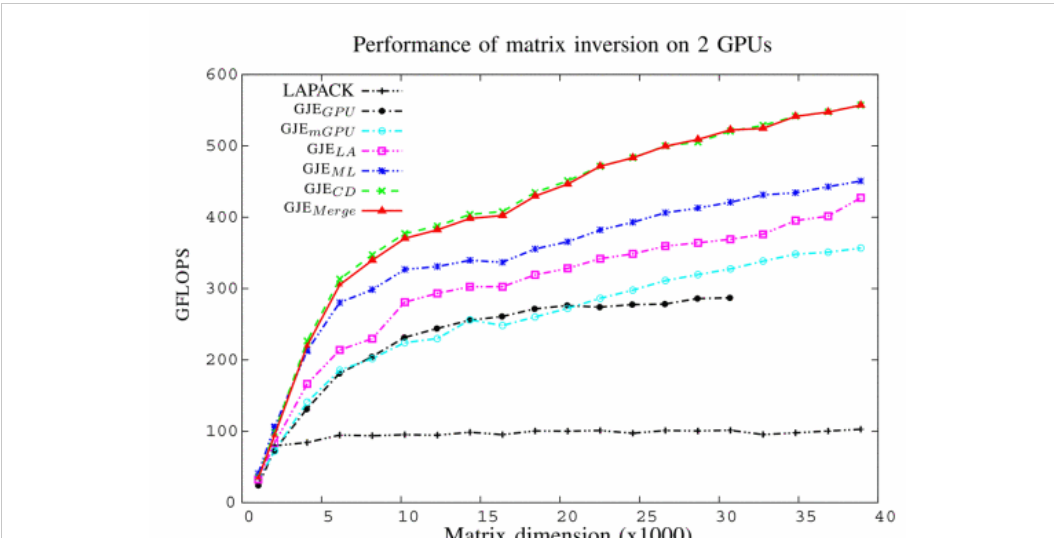
TABLE I HARDWARE EMPLOYED IN THE EXPERIMENTS.

Processors	#processors	#cores (per proc.)	Frequency (GHz)	L2 cache (MB)	Memory (GB)	Single precision peak performance (GFLOPs)
Intel Xeon QuadCore E5530	2	4	2.27	8	48	(2×)76.8
NVIDIA TESLA c1060	4	240	1.3	-	(4×4)16	(4×)622.0

Figure 8 shows the performance (in GFLOPS) attained by the best implementation,  $GJE_{Merge}$ , when running on 1, 2, 3 and 4 GPUs for large problem sizes. The results in this figure illustrate the scalability of the new implementation. Note that, as the number of GPUs is increased, the available memory also grows and, in consequence, we can address problems of larger dimension.

SECTION V.  
Related Work

There has been much recent work exploring the benefits of dynamic scheduling of dense linear algebra operations on multi-core processors; visit, e.g. the web sites of the FLAME (<http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage>), PLASMA (<http://icl.cs.utk.edu/plasma/>) and SMPs ([http://www.bsc.es/plantillaG.php?cat\\_id=385](http://www.bsc.es/plantillaG.php?cat_id=385)). This research was originally modified to address multi-GPU platforms in [9], and later followed by [11], [12], [13] and StarPU [14]. Although in principle this approach can also be applied to matrix inversion, to effectively increase performance, dynamical scheduling has to be combined with a modified scheme of pivoting known as incremental pivoting [10], different from the traditional partial pivoting used in Gaussian elimination and GJE. Incremental pivoting is known to yield an increase of element growth during the factorization and, therefore, is considered to be numerically inferior to partial pivoting. In addition to this, efficient implementation of matrix factorization/inversion via an algorithm-by-blocks with incremental pivoting requires the combination of two block sizes and certain kernels which are far from being efficiently implemented on the GPU. Finally, although matrix inversion via the traditional approach, i.e., LU factorization with partial pivoting, is also possible, this would require a kernel for inversion of a triangular matrix, an operation which is not available in current libraries for GPUs.



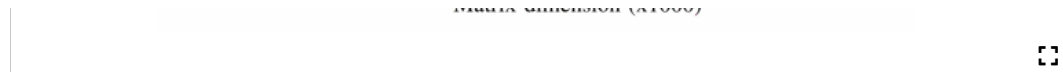


Fig. 6. Performance of the different variants for matrix inversion on 2 GPUS.

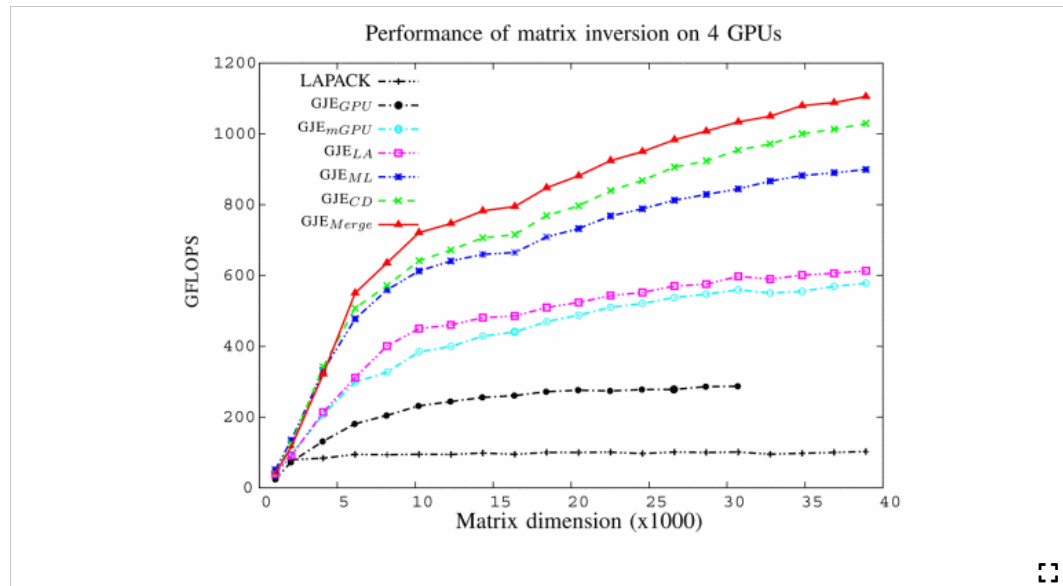


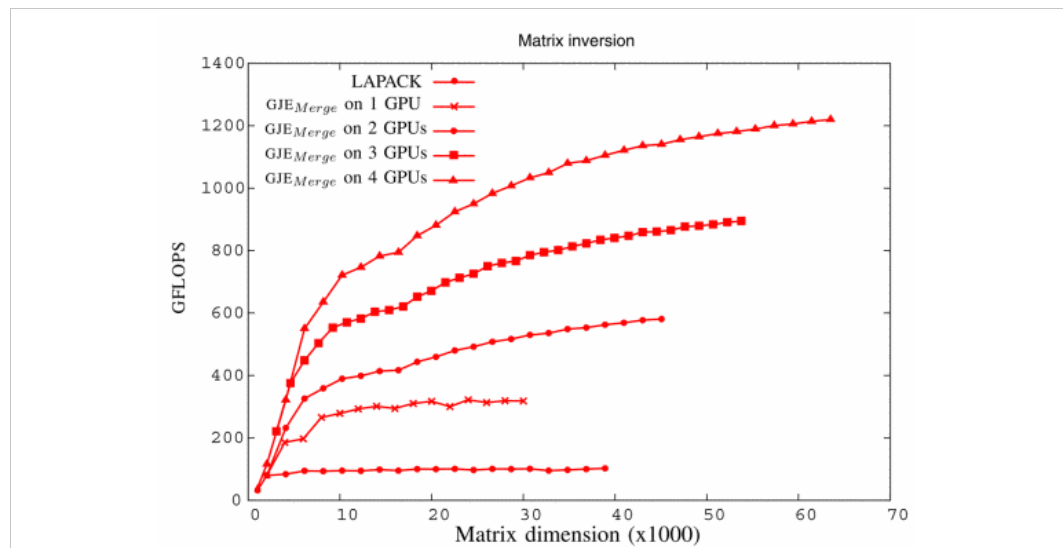
Fig. 7. Performance of the different variants for matrix inversion on 4 GPUS.

## SECTION VI.

### Concluding Remarks And Future Work

We have presented five different hybrid variants for matrix inversion based on GJE. The target platform consists of a multi-core processor connected to multiple GPUs. The implementations exploit the capabilities of the underlying platform, executing each operation on the most convenient device and computing concurrently on all the available computational units.

The initial implementation,  $GJE_{mGPU}$ , is based on a hybrid approach for a single GPU described at [3]. Results on a platform with 4 GPUs show that  $GJE_{mGPU}$  is nearly six times faster than the LAPACK implementation, but its performance is still low. The rest of the implementations are the result of an incremental refinement of  $GJE_{mGPU}$ . The best version,  $GJE_{Merge}$ , is two times faster than  $GJE_{mGPU}$  on 4 GPUs, and exhibits excellent scalability properties (with a nearly linear speed-up).





**Fig. 8.** Performance of variant  $GJE_{Merge}$  for the GJE-based matrix inversion on 1, 2, 3 and 4 GPUs.

We find two positive effects from the use of multiple GPUs: a considerable reduction of the computational time and an increase of the available memory, allowing the inversion of larger matrices.

Future research resulting from this work will consider:

- The performance of  $GJE_{Merge}$  is clearly limited by the performance of the CUBLAS routine for matrix-matrix products. Other GPU kernels should be evaluated.
- Although performance in double-precision is considerably slower than in single-precision for many GPUs, NVIDIA Fermi products drastically reduce this difference. Thus, it is worth evaluating the impact of this improvement on the performance of double-precision matrix inversion.

Acknowledgement

Authors

Figures

References

Citations

By PapersBy Patents

Download PDFs

Export ?

References & Cited By

Cites in Papers - IEEE (5) | Other Publishers (6)

Cites in Papers - IEEE (5)

☐ Select All

☐ 1.

Bahar Asgari, Dheeraj Ramchandani, Amaan Marfatia, Hyesoon Kim, "Maia: Matrix Inversion Acceleration Near Memory", *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pp.277-281, 2022.

Show Article Google Scholar

☐ 2.

Jin-Bae Park, Kwang Soon Kim, "Load-Balancing Scheme With Small-Cell Cooperation for Clustered Heterogeneous Cellular Networks", *IEEE Transactions on Vehicular Technology*, vol.67, no.1, pp.633-649, 2018.

Show Article Google Scholar

☐ 3.

Ryma Mahfoudhi, Sami Achour, Olfa Hamdi-Larbi, Zaher Mahjoub, "High Performance Recursive Matrix Inversion for Multicore Architectures", *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pp.675-682, 2017.

Show Article Google Scholar

☐ 4.

Jun Liu, Yang Liang, Nirwan Ansari, "Spark-Based Large-Scale Matrix Inversion for Big Data Processing", *IEEE Access*, vol.4, pp.2166-2176, 2016.

Show Article Google Scholar

☐ 5.


Alejandro López-Ortiz, Alejandro Salinger, Robert Suderman, "Toward a Generic Hybrid CPU-GPU Parallelization of Divide-and-Conquer Algorithms", *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp.601-610, 2013.

Show Article  Google Scholar 

Cites in Papers - Other Publishers (6)


1.

Santiago Moreno-Carbonell, Eugenio F. Sánchez-Úbeda, "A Piecewise Linear Regression Model Ensemble for Large-Scale Curve Fitting", *Algorithms*, vol.17, no.4, pp.147, 2024.

CrossRef Google Scholar 


2.

Chandan Misra, Swastik Haldar, Sourangshu Bhattacharya, Soumya K. Ghosh, "SPIN", *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pp.1, 2018.

CrossRef Google Scholar 


3.

M. Varalakshmi, Amit Parashuram Kesarkar, Daphne Lopez, "Embarrassingly Parallel GPU Based Matrix Inversion Algorithm for Big Climate Data Assimilation", *International Journal of Grid and High Performance Computing*, vol.10, no.1, pp.71, 2018.

CrossRef Google Scholar 


4.

Alejandro López-Ortiz, Alejandro Salinger, Robert Suderman, "Toward a Generic Hybrid CPU-GPU Parallelization of Divide-and-Conquer Algorithms", *International Journal of Networking and Computing*, vol.4, no.1, pp.131, 2014.

CrossRef Google Scholar 


5.

Peter Benner, Pablo Ezzatti, Hermann Mena, Enrique Quintana-Orti, Alfredo Remon, "Solving Matrix Equations on Multi-Core and Many-Core Architectures", *Algorithms*, vol.6, no.4, pp.857, 2013.

CrossRef Google Scholar 

6.

I. Demir, R. Westermann, "Progressive High-Quality Response Surfaces for Visually Guided Sensitivity Analysis", *Computer Graphics Forum*, vol.32, no.3pt1, pp.21, 2013.

CrossRef Google Scholar 

Keywords 





Metrics 


More Like This

Speedup of Implementing Fuzzy Neural Networks With High-Dimensional Inputs Through Parallel Processing on Graphic Processing Units  
IEEE Transactions on Fuzzy Systems  
Published: 2011

A Survey of Caching Techniques for General Purpose Graphics Processing Units  
2024 3rd International Conference for Innovation in Technology (INOCON)  
Published: 2024

Show More

IEEE Personal Account	Purchase Details	Profile Information	Need Help?	Follow
CHANGE USERNAME/ PASSWORD	PAYMENT OPTIONS VIEW PURCHASED DOCUMENTS	COMMUNICATIONS PREFERENCES PROFESSION AND EDUCATION	US & CANADA: +1 800 678 4333  WORLDWIDE: +1 732 981 0060	   

[About IEEE Xplore](#) | [Contact Us](#) | [Help](#) | [Accessibility](#) | [Terms of Use](#) | [Nondiscrimination Policy](#) | [IEEE Ethics Reporting](#)  | [Sitemap](#) | [IEEE Privacy Policy](#)

A public charity, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2024 IEEE - All rights reserved, including rights for text and data mining and training of artificial intelligence and similar technologies.

## IEEE Account

» [Change Username/Password](#)

» [Update Address](#)

## Purchase Details

» [Payment Options](#)

» [Order History](#)

» [View Purchased Documents](#)

## Profile Information

» [Communications Preferences](#)

» [Profession and Education](#)

» [Technical Interests](#)

## Need Help?

» **US & Canada:** +1 800 678 4333

» **Worldwide:** +1 732 981 0060

» [Contact & Support](#)

[About IEEE Xplore](#) | [Contact Us](#) | [Help](#) | [Accessibility](#) | [Terms of Use](#) | [Nondiscrimination Policy](#) | [Sitemap](#) | [Privacy & Opting Out of Cookies](#)

A not-for-profit organization, IEEE is the world's largest technical professional organization dedicated to advancing technology for the benefit of humanity.

© Copyright 2024 IEEE - All rights reserved. Use of this web site signifies your agreement to the terms and conditions.