Comput Sci Res Dev (2016) 31:215–223
DOI 10.1007/s00450-016-0326-3

SPECIAL ISSUE PAPER

Energy-aware solution of linear systems with many right hand sides

Martin Köhler[1] · **Carolin Penke**[1] · **Jens Saak**[1] · **Pablo Ezzatti**[2]

Published online: 12 August 2016

Abstract The solution of linear systems of equations with many right hand sides is mostly seen as a trivial extension of solving a linear system and the algorithmic developments mostly focus on the efficient computation of the LU decomposition. This is, however, not regarding the case where many right hand sides increase the runtime influence of the forward/backward substitution. In this contribution we present a GPU accelerated Gauss–Jordan-elimination based all-at-once solution scheme which focuses on minimizing the runtime and the energy consumption by switching the forward/backward substitution in favor of a more suitable operation. We obtain a multi-GPU aware algorithm which is up to 2.5 times faster than the current state-of-the-art LU decomposition based solution process of MAGMA and saves 48 % required energy.

Keywords Linear systems · GPU computing · Gauss–Jordan-elimination · Energy-awareness

Mathematics Subject Classification 65F05

✉ Martin Köhler
koehlerm@mpi-magdeburg.mpg.de

Carolin Penke
penke@mpi-magdeburg.mpg.de

Jens Saak
saak@mpi-magdeburg.mpg.de

Pablo Ezzatti
pezzatti@fing.edu.uy

[1] Computational Methods in Systems and Control Theory, Max-Planck-Institute for Dynamics of Complex Technical Systems, Sandtorstr. 1, 39106 Magdeburg, Germany

[2] Facultad de Ingenrieríra, Universidad de la República, Julio Herrera y Reissig 565, Código Postal 11.300 Montevideo, Uruguay

1 Introduction

The solution of linear systems is still one of the most common operations in scientific computing. The efficient solution of these systems is a key ingredient for many higher level algorithms. In our contribution we focus on the solution of

$$AX = B, \tag{1}$$

where $A \in \mathbb{R}^{m\times m}$ is a non-symmetric matrix, $B \in \mathbb{R}^{m\times n}$, and $X \in \mathbb{R}^{m\times n}$ with $n \gg 1$ without any special structure. Especially, the case $m = n$, which appears in the computation of the generalized matrix sign function [5], is of higher interest. The standard implementation for this problem is using the LU decomposition with an additional forward/backward substitution. Regarding current GPU accelerated implementations, where we consider the MAGMA [2,9,10] version the state-of-the-art, only the LU decomposition works on more than one accelerator device. This is a crucial point to reduce the runtime and as a consequence also the energy consumption.

A closer look to the LU decomposition based solution process shows an additional problem: The three step scheme factorizing the matrix $PA = LU$, followed by the forward solve $LY = PB$, and the backward solve $UX = Y$, leads to the matrix being transferred between the main memory and the computational unit for three times. Since memory accesses are a key player in the energy balance, this also results in a notable additional power consumption.

In order to be able to not only factorize the matrix on multiple accelerators but also solve the linear system without the communication intensive distributed forward–backward substitution we recall the Gauss–Jordan-elimination algorithm. In contrast to the LU decomposition, the integration of the right hand side in the computation scheme removes the data

dependencies which appear in the forward/backward substitution scheme. This enables us to derive an easy and efficient multi-GPU implementation of the solution process.

Beside focusing on the classical objective of minimizing the time-to-solution, we also take the energy-to-solution into account. Regarding the hardware developments during the recent years, an increasing number of the scientific computing hardware has been set up as heterogeneous architectures. Most commonly the combination of general purpose CPUs together with GPU-based accelerator devices is used. These architectures allow to choose the computation device that is best suited for a given task with respect to either the time-to-solution, or in the green HPC context, combinations with the energy-to-solution. The aforementioned differing properties of the LU decomposition and the Gauss–Jordan-elimination, with respect to saving memory transfers and the usage of multiple accelerators for the overall process, define the objective to look for obtaining the method that performs best with respect to both the time and the energy metric when we want to solve linear systems with many right hand sides.

In the following sections we recall the Gauss–Jordan-elimination and reformulate it to work on heterogeneous system with multiple accelerator devices. Furthermore, we identify and avoid bottlenecks of the GPU based implementation, which slow down the computation on the one hand, but are keeping the accelerator devices busy and consuming energy on the other hand. Finally, the numerical experiments compare our Gauss–Jordan based solver with the LU decomposition from MAGMA library [2,9,10], which can be seen as the GPU accelerated implementation of LAPACK. These experiments focus on the runtime as well as on the energy consumption of both algorithms and the multi-accelerator scalability of the Gauss–Jordan approach and the LU decomposition based solution process.

2 Gauss–Jordan-elimination

The Gauss–Jordan-elimination (GJE) process is mostly known as an algorithm to invert matrices [1,7]. Obviously, inverting the system matrix A first and multiplying the right hand side B by A^{-1} afterwards is a quite expensive way to solve a linear system. It requires more than $2m^3 + 2m^2n$ flops. However, the involved matrix-matrix products can be easily performed in parallel on multiple accelerator devices. On the other hand, the typical way of calculation using the LU decomposition followed by a forward/backward substitution only costs $\frac{2}{3}m^3 + 2m^2n$ flops. The data dependency caused by the L and the U factors increase the communication necessary during the solution phase. The Gauss–Jordan-elimination scheme derived in the next paragraphs will reduce (when comparing the inversion and the matrix multiplication) the number of flops to only $m^3 + 2m^2n$, for the solution of one linear system, without introducing data dependencies that break the easy use of multiple accelerator devices.

2.1 Basic scheme

We consider the *augmented matrix*

$$D := [\ A\ \mid B\ ] = \left[\begin{array}{ccc|ccc} a_{11} & \cdots & a_{1m} & b_{11} & \cdots & b_{1n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & \cdots & a_{mm} & b_{m1} & \cdots & b_{mn} \end{array}\right] \tag{2}$$

of size $m \times (m+n)$, which concatenates the system matrix A and the right hand side B. Now, the goal of the classic Gauss–Jordan-elimination is to transform the first m columns of the matrix D to the identity using elementary operations from the left, while the last n columns, initially containing the right hand side, are overwritten by the solution X, simultaneously. We use the formulation with column pivoting. Therefore, for column $i \in \{1, \ldots, m\}$ of A, every elementary operation consists of a row permutation P_i which exchanges the i-th and the k-th row ($k \geq i$ such that a_{ki} has the largest absolute value) and a Gauss transformation

$$G_i = \begin{bmatrix} 1 & & -\frac{a_{1i}}{a_{ii}} & & \\ & \ddots & \vdots & & \\ & & 1 & -\frac{a_{(i-1)i}}{a_{ii}} & \\ & & & \frac{1}{a_{ii}} & \\ & & -\frac{a_{(i+1)i}}{a_{ii}} & 1 & \\ & & \vdots & & \ddots \\ & & -\frac{a_{mi}}{a_{ii}} & & 1 \end{bmatrix}. \tag{3}$$

creating a one at the new (i, i) position in A, or D. Thus, we end up with

$$\underbrace{G_n P_n \cdots G_2 P_2 G_1 P_1}_{A^{-1}} D = \left[\begin{array}{ccc|ccc} 1 & & & x_{11} & \cdots & x_{1n} \\ & \ddots & & \vdots & \vdots & \vdots \\ & & 1 & x_{m1} & \cdots & x_{mn} \end{array}\right], \tag{4}$$

which gives us A^{-1} and X with a cost of $2m^3 + 2m^2n$ flops.

2.2 Block reformulation

Since it is well known that only BLAS level-3 enabled algorithms are able to get near the maximum theoretic performance of the computation device we reformulate the above idea into a blocked algorithm. To this end, we first rewrite the update (3) into a rank-1 update [7]:

$$D := D - \frac{1}{a_{ii}}\left(a_{1i}, \ldots, a_{(i-1)i}, 0, a_{(i+1)i}, \ldots, a_{mi}\right)^T D_{i,\cdot}$$
$$D_{i,\cdot} := \frac{1}{a_{ii}} D_{i,\cdot}.$$

Afterwards, we repartition the augmented matrix D into

$$D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & b_1 \\ \hline A_{21} & A_{22} & A_{23} & b_2 \\ \hline A_{31} & A_{32} & A_{33} & b_3 \end{array}\right], \tag{5}$$

where A_{22} is of dimension $N_B \times N_B$. In this way the rank-1 update is transformed into a rank-N_B update by replacing the scalar operations by their corresponding matrix valued counterparts:

$$D := \left[\begin{array}{c|c|c|c} A_{11} & 0 & A_{13} & b_1 \\ \hline 0 & 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} & b_3 \end{array}\right] + \left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right] \left[\begin{array}{c} A_{21}^T \\ I_{N_B} \\ A_{23}^T \\ b_2^T \end{array}\right]^T, \tag{6}$$

In order to preserve numerical stability, we have to lift the pivoting strategy to the block reformulation, as well [7]. This can be done either by implementing an unblocked Gauss–Jordan-elimination as shown in (4) for the panel $\left[A_{12}^T\ A_{22}^T\ A_{32}^T\right]^T$, or by computing the pivoted LU decomposition of $\left[A_{22}^T\ A_{32}^T\right]^T$. The later leads to

$$P \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} U_1. \tag{7}$$

The obtained permutation is applied to the augmented matrix D by

$$D := \begin{bmatrix} I_{11} & \\ & P \end{bmatrix} D,$$

where I_{11} is the identity matrix of the same size as A_{11} in the partitioning (5). By introduction of the panel matrix H

$$H = \begin{bmatrix} -A_{12}U_1^{-1}L_1^{-1} \\ U_1^{-1}L_1^{-1} \\ -L_2L_1^{-1} \end{bmatrix}$$

the Update (6) transforms into

$$D := \left[\begin{array}{c|c|c|c} A_{11} & 0 & A_{13} & b_1 \\ \hline 0 & 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} & b_3 \end{array}\right] + H\left[A_{21}\ I_{N_B}\ A_{23}\ b_2\right]. \tag{8}$$

This procedure will compute the solution of the Linear System (1) as well as the inverse of A. Avoiding the left side of the update corresponding to the block A_{21}, we only compute the solution of the linear system. The resulting overall procedure using this second alternative, is shown in Algorithm 1. By only computing the solution of the linear system, we need $m^3 + 2m^2n$ flops to solve a linear system. The negligible influence of the block size N_B and its determination is shown in [3]. As already pointed out in the introduction, we observe that the Gauss–Jordan-elimination scheme only needs to sweep over the matrix A once instead of three times of the LU decomposition based solution.

Algorithm 1 Solution of linear systems using Gauss–Jordan-elimination

Input: $A \in \mathbb{R}^{m\times m}$ nonsingular, $B \in \mathbb{R}^{m\times n}$
Output: $X \in \mathbb{R}^{m\times n}$ solving $AX = B$ overwriting B
1: Set $D := \left[A\ B\right] \in \mathbb{R}^{m\times m+n}$.
2: **for** $J := 1, 1+N_B, 1+2N_B, \ldots, n$ **do**
3: $J_B := \min(N_B, n - J + 1)$
4: Partition $D := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & A_{13} & B_1 \\ \hline A_{21} & A_{22} & A_{23} & B_2 \\ \hline A_{31} & A_{32} & A_{33} & B_3 \end{array}\right]$,
where $A_{11} \in \mathbb{R}^{J-1\times J-1}$ and $A_{22} \in \mathbb{R}^{J_B\times J_B}$.
5: $P_i^T \left[\frac{L_1}{L_2}\right] U_1 := \left[\frac{A_{22}}{A_{32}}\right]$, update $P := P_i P$. {GETRF}
6: Permute $\left[\frac{A_{23}|B_2}{A_{33}|B_3}\right] := P_i \left[\frac{A_{23}|B_2}{A_{33}|B_3}\right]$. {LASWP}
7: Set $H := \left[\begin{array}{c} -A_{12}U_1^{-1}L_1^{-1} \\ \hline U_1^{-1}L_1^{-1} \\ \hline -L_2L_1^{-1} \end{array}\right]$. {TRSM}
8: and $\left[\frac{A_{13}|B_1}{A_{23}|B_2}{A_{33}|B_3}\right] := \left[\frac{A_{13}|B_1}{0|0}{A_{33}|B_3}\right] + H\left[A_{23}|B_2\right]$. {GEMM}
9: **end for**

2.3 Distributed algorithm

The previous section showed that beside calculating the current panel matrix H, we only need the GEMM operation to solve the linear system. This induces that inside a column or a block column the only data dependency is the knowledge of the current panel H. Once this is known all columns can be updated independent from each other. This motivates to distribute in a cyclic block-column way across all participating computational devices. Algorithm 1 only needs to distribute the augmented matrix D over all computational devices and to broadcast the current panel matrix H in every iteration. The GEMM calls for the permutation updating the remaining part of the matrix and the right hand side can be performed in parallel on all computational devices. At this point we have to remark that the cyclic block-column distribution is only efficient if the number of computational device is relatively small. Normally this should not affect our idea because the maximum number of accelerator devices inside one compute server is relatively small ($\leq$8) by nature.

2.4 Memory access analysis

In this subsection we count the number of memory accesses required by the LU decomposition and the GJE under some simplifications due to the complexity of modern hardware. We assume that only scalar values stay inside the cache of the CPU or GPU. By increasing the dimension of the problems matrices and vectors one can make them large enough to no longer fit into caches. Finally, we assume both algorithms, the LU decomposition and the GJE, use only rank-1 updates without pivoting, i.e., we regard their BLAS level-2 formulation.

Each step k in the LU decomposition consists of a vector scaling and a rank-1 update. The vector scaling needs $1 + k$ memory reads and k writes. Processing the rank-1 update row-by-row, each row needs $2k + 1$ reads and k writes. For $k = m - 1, \ldots, 1$ the overall LU decomposition needs

$$\sum_{k=1}^{m} 3k^2 + 2k + 1 = m^3 - \frac{1}{2}m^2 + \frac{1}{2}m - 1$$

memory accesses. A forward (or backward) substitution for one column of the right hand side needs

$$\sum_{i=1}^{m} \left[3 + \sum_{j=1}^{i-1} 4 \right] = 2m^2 + m$$

memory accesses. Together with the assumption from the introduction ($n = m$) this yields

$$5\,m^3 + \frac{3}{2}m^2 + \frac{1}{2}\,m - 1 \tag{9}$$

memory accesses to solve a linear system with m right hand sides.

The whole Gauss–Jordan-elimination scheme consists only of $m - 1$ vector scales of length m and $m - 1$ rank-1 updates of size $m \times (k + n)$ where $k = m - 1, \ldots, 1$. Each update costs $(3(k+n)+1)m$ memory accesses. Again, together with the assumption $n = m$, this gives

$$\sum_{k=1}^{m-1} m + (3(k + m) + 1)m = \frac{9}{2}m^3 - \frac{5}{2}m^2 - 2m \tag{10}$$

memory accesses. Compared to the LU decomposition this saves the transfer of $\frac{1}{2}m^3$ elements.

3 GPU implementation

The GPU implementation splits into two parts. First, we introduce a look-ahead scheme in order to introduce parallelism between host CPU and the accelerator device. Second, we discuss reasons for changing from the classical Fortran column major matrix storage to the row major storage known from C.

Beside the high computational power of the accelerator devices, a work sharing between CPU and GPU is a key ingredient for reducing the runtime. Therefore, we divide the operation performed by Algorithm 1 into two classes, those well suited for the GPU and those to be run on the host CPU(s). The operations well suited for the GPU are the GEMM operations, benefiting from the high computational power of the GPU, and the row swap of the pivoting, because of the high memory throughput of the GPU. The preparation of the panel matrix H is better suited for the CPU, due to the lower number of required operations in general and large sequential parts. Copying the current panel $\left[A_{12}^T\ A_{22}^T\ A_{23}^T\right]^T$ to the CPU, computing H moving it back to the GPU, results in a simple GPU accelerated implementation of Algortihm 1.

3.1 Look-ahead and asynchronous operation

In the basic GPU accelerated algorithm only the CPU or the GPU will work at a time. Since the GPUs are able to transfer data between the host and their memory while performing computation, we introduce a classic look-ahead strategy with the aim to prepare the next panel $\tilde{H}$ on the CPU, while the GPU still updates the remaining parts of the augmented matrix D. Therefore, we split the third column of the block partitioning (5) into

$$\begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix} := \begin{bmatrix} \hat{A}_{13} & \bar{A}_{13} \\ \hat{A}_{23} & \bar{A}_{23} \\ \hat{A}_{33} & \bar{A}_{33} \end{bmatrix},$$

where $\hat{A}_{13}$, $\hat{A}_{23}$, and $\hat{A}_{33}$ have N_B columns. This small additional repartitioning allows us to update $\hat{A}_{13}$, $\hat{A}_{23}$, and $\hat{A}_{33}$ first and copy them back to the host while the GPU performs the remaining updates on $\bar{A}_{13}$, $\bar{A}_{23}$, and $\bar{A}_{33}$. This way, the host prepares the next panel matrix $\tilde{H}$, while the GPU still works on the previous update. The new panel matrix $\tilde{H}$ is potentially copied back to the device in the same asynchronous while the updates are still ongoing. After the updates on $\bar{A}_{13}$, $\bar{A}_{23}$, and $\bar{A}_{33}$ we synchronize the computations done on the device to ensure that the updated $\tilde{H}$ has been tranferred completely before it is used.

3.2 Data layout

Due to the fact that the BLAS libraries available for GPUs, namely NVIDIA® cuBLAS for CUDA enabled devices and clBLAS for OpenCL based devices, use the same matrix storage scheme as the CPU based libraries BLAS and LAPACK

Table 1 Portion of the row-swap operations at the solution time on a single NVIDIA® Tesla K20 and with a block size of $N_B = 1024$ (double precision)

Dimension		Swap	Swap + transpose
m	n	Column major (%)	Row major (%)
5120	1	32.5	14.9
	5120	37.5	8.6
10,240	1	24.8	7.2
	10,240	24.4	5.0
15,360	1	21.7	4.6
	15,360	20.0	3.7

we implement Algorithm 1 using column major storage. Thereby, preliminary experiments showed that 20–37.5 % of the computation time on the GPU (NVIDIA® Tesla K20) is spent in applying the permutation P to the remaining parts of the matrix (Step 2.2 of Algorithm 1). Previous work on the Gauss–Jordan-elimination based solvers [3] is affected by this issue. From this implementation we obtained Table 1 showing the percentage of time used for applying the permutation P.

A thorough analysis of the operation shows that permuting rows, despite being an easy operation in terms of the column major storage scheme, disturbs the coalesced memory access scheme of accelerator device. The reason is that for each row swap at most two elements are used out of each cache line. Assuming a cache line length of 128 bytes, which is typically used on a NVIDIA® CUDA device, and double precision arithmetic, we only use 8 or 16 bytes out of a cache line, which means that 93.5 or 87.5 % of the data transferred from the memory is not used, while its transfer is wasting time and energy, and slows down the overall process. Regarding the length of a cache line, it would be better if we could store 16 elements of a row in one cache line such that exchanging 16 elements of two rows only requires a transfer of 256 bytes from the memory, instead of 4kB. The direct consequence of this fact is that on the accelerator device the row major storage scheme is better suited for row swaps. This requires that at least the part of the algorithm working on the device needs to be adjusted to row major storage. Beside some copy operations this only affects the GEMM operation.

We assume that a call to `GEMM(`α`, A, B,` β`, C)` computes $C := \alpha AB + \beta C$, where $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, and $C \in \mathbb{R}^{m\times n}$ stored in (Fortran) column major order. Furthermore, it is obvious that A^T in column major storage is the same as A in row major storage. It follows that the column major oriented routine `GEMM(`α`, B, A,` β`, C)` then computes

$$C^T := \alpha B^T A^T + \beta C^T,$$

if the matrices are stored in row major, which gives our original GEMM operation in column major storage. Furthermore, that means that we only have to switch the dimension arguments and the roles of A and B for using the classical GEMM operations with the row major scheme. The numerical experiments show that this reduces the influence of row swap operation. The transpose operation which is additionally required is only performed after the initial transfer to the device and before the final results are copied back to the host. Therefore, they can be neglected in the performance analysis as the numerical results show.

4 Experimental results

In the experimental results we will compare our Gauss–Jordan-elimination based approach with the well established GPU accelerated implementation of the LU decomposition from the runtime as well as from the energy point of view. Therefore, we choose random matrices $A \in \mathbb{R}^{m\times m}$ with $m = 1024k$, $k \in \mathbb{N}$, and a predefined solution $X \in \mathbb{R}^{m\times n} = 1$. From this we determine the right hand side as $B = AX$. We set n equal to m in order to cover the case we explained in the introduction. All experiments are done in double precision arithmetic on a dual-socket 16 core Intel® Xeon® E5-2640v3 with 64GB RAM and two NVIDIA® Tesla K20 accelerators. The code is compiled using the Intel® C/Fortran compiler 15 with MKL 11.2 as host BLAS library. The device code uses NVIDIA® CUDA 7.5. The power consumption is measured using a ZES Zimmer LMG450 power meter with a sampling rate of 20 Hz.

Because of the fact, that reducing the power consumption results in a slower execution, i.e. larger runtime, in many cases, we consider the *energy-delay-product (EDP)* [4,6] as a combined economical and ecological measure to compare both solution techniques. The energy-delay-product is defined as

$$\mathrm{EDP}(w) = E \cdot T^w, \tag{11}$$

where E is the energy-to-solution, T is the time-to-solution and w a weight factor to penalize the time. Usually, the EDP(1), EDP(2), and EDP(3) values are used for the comparison of algorithms depending on the requirements of the computing center.

Due to the fact that MAGMA [2,9,10] supports the LU decomposition on multiple devices, but the forward/backward substitution on a single device only, we use a workaround here. In the case of one GPU we use the MAGMA LU decomposition and the corresponding forward/backward solve on this device. In the case where we use both accelerator devices we only use the multi-GPU LU decomposition and perform the forward/backward substitution on the host CPU.

Table 2 Optimal block size N_B for minimizing time and energy-to-solution on one GPU (time in [s], energy in [Ws])

$m = n$	Energy optimal			Time optimal		
	N_B	Energy	Time	N_B	Time	Energy
2048	256	23.23	0.06	256	0.06	23.23
4096	512	123.49	0.32	640	0.31	124.33
6144	768	378.11	0.91	1024	0.90	379.56
8192	1024	854.20	2.00	1280	1.98	854.56
10,240	1152	1608.78	3.73	1536	3.68	1614.10
12,288	896	2614.01	6.34	1536	6.29	2680.57
14,336	1408	3896.29	9.73	2048	9.53	3936.05
16,384	1024	5568.58	14.36	1408[a]	14.17	5578.11

[a] Restricted to 1408 by device memory

Table 3 Energy-delay-product ($w = 1$) of the optimal block sizes using one GPU

$m = n$	8192	10,240	12,288	14,336	16,384
Energy opt.	1708.4	6000.7	16,572.8	37,910.9	79,964.8
Time opt.	1692.0	5939.9	16,860.8	37,510.6	79,041.8

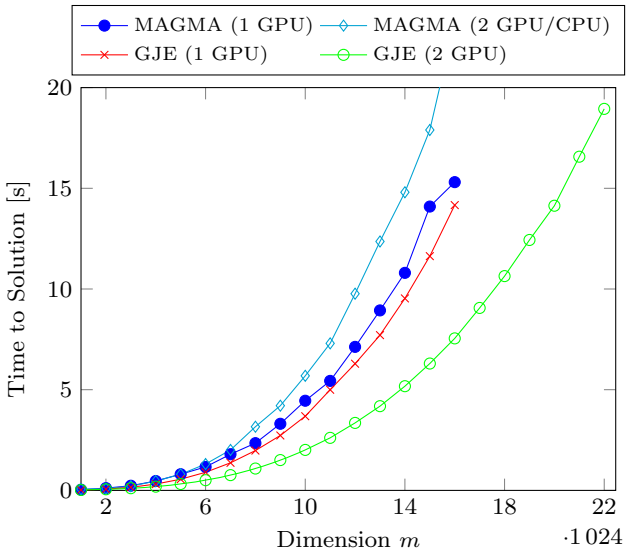


Fig. 1 Time-to-solution ($n = m$)

Table 4 Runtime (in [s]) of small and medium size problem, speed up against single GPU MAGMA

$m = n$	MAGMA	GJE: 1 GPU		GJE: 2 GPU	
	Time	Time	Speed up	Time	Speed up
1024	0.03	0.02	1.50	0.02	1.50
2048	0.11	0.06	1.83	0.06	1.83
3072	0.23	0.15	1.53	0.11	2.09
4096	0.47	0.31	1.52	0.19	2.47
5120	0.80	0.56	1.43	0.32	2.50
6144	1.17	0.90	1.30	0.51	2.29
7168	1.80	1.37	1.31	0.76	2.37
8192	2.35	1.98	1.19	1.09	2.15

Before we compare the Gauss–Jordan-elimination approach with the LU decomposition based solvers, we have to determine the optimal block size N_B with respect to the solution time and with respect to the energy, which are shown in Table 2. For the decision which optimal value to choose for the experiments, we consider the energy-delay-product shown in Table 3. Except of one case the EDP(1) suggest to chose the time optimal block size, which is therefore used for all remaining experiments.

Figure 1 shows the time-to-solution for all methods in the test. Obviously the usage of MAGMA with more than one GPU and the additional triangular solves on the CPU yield the worst result. Neither from the run-time nor energy points of view can it gain any advantage. Restricting to the one GPU case the Gauss–Jordan-elimination approach gains a speed up against MAGMA although we need 12.5 % more flops to solve. For small problems a speed up of more than 1.5, as shown in Table 4, is possible but even for the large problems our approach is 8 % faster. Regarding the energy-to-solution in Table 5 our approach saves between 8 and 60 % energy for the small and moderate size problems. In the case of the large problems, we see in Fig. 2 that we need approximately the same amount of energy to compute the solution. Because of minimizing the runtime with keeping the energy constant the energy-delay-product in Table 6 suggest to choose the Gauss–Jordan-elimination. Even increasing the weight w will not change this picture due to the fact that the runtime speed up gets more influence in the assessment.

Employing two GPUs we see in Table 4 that our distributed Gauss–Jordan-elimination implementation is the fastest approach for problems beginning at medium size ($m = n \geq 3\,072$). The overall speed up with respect to one GPU MAGMA solution is between 2.09 and 2.5. Furthermore, comparing with our one GPU implementation, we see a nearly perfect scaling. This nearly perfect scaling yield a large runtime reduction which easily compensates the additional power necessary for the second GPU. Even in the cases where no difference in runtime exists between the one and the two GPU execution the two GPU version requires less energy than MAGMA on one GPU. The direct comparison between the one and the two GPU implementation shows that using a second device accelerates the computation by a factor of up to 2 and reduces the energy consumption by 22 %. Regarding the energy-delay-product again in Table 6 it suggests to use the two GPU variant if two GPUs are available.

Reasons for the Gauss–Jordan-elimination scheme being so much more efficient than the LU decomposition with forward/backward substitution are the following:

Table 5 Energy-to-solution (in [Ws]) of small and medium size problem, savings against single GPU MAGMA

$m = n$	MAGMA	GJE: 1 GPU		GJE: 2 GPU	
	Energy	Energy	Savings (%)	Energy	Savings (%)
1024	6.45	2.64	59.07	3.35	48.06
2048	37.75	23.23	38.46	20.72	45.11
3072	85.23	62.87	26.24	46.13	45.88
4096	171.60	124.33	27.55	85.20	50.35
5120	288.23	227.49	21.07	150.72	47.71
6144	415.02	379.56	8.55	253.48	38.92
7168	639.03	585.90	8.31	393.24	38.46
8192	845.31	854.56	−1.09	586.76	30.59

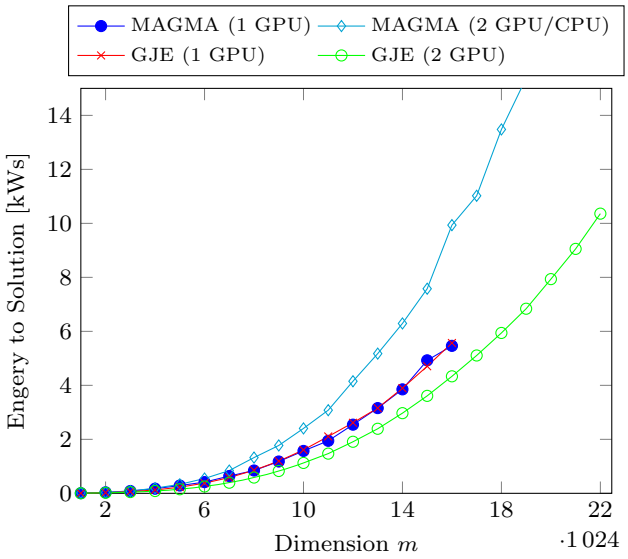


Fig. 2 Energy-to-solution ($n = m$)

Table 6 Energy-delay-product ($w = 1$)

$m = n$	MAGMA 1 GPU	MAGMA 2 GPU/CPU	GJE 1 GPU	GJE 2 GPU
2048	4.3	2.7	1.4	1.2
4096	80.0	88.5	38.8	15.8
6144	483.6	718.1	343.5	128.4
8192	1983.6	4184.7	1704.8	639.5
10,240	6990.3	13,678.1	6001.7	2262.0
12,288	18,123.5	40,545.6	16,578.6	6399.3
14,336	41,601.4	93,147.9	37,919.4	15,364.3
16,384	83,609.4	230,820.8	79,985.4	32,713.2
18,432	–	430,949.8	–	63,212.7
20,480	–	829,642.6	–	114,236.3
22,528	–	1,355,716.0	–	196,836.7

– Gauss–Jordan is an all-at-once approach where one algorithm and one sweep over the augmented matrix D yield the solution of the problem. In comparison to the LU decomposition this reduces the memory transfers, since no additional triangular solves are required.
– The GEMM operation is the only operation necessary on the device. It does not involve data dependencies which yield partly sequential computation schemes, in contrast to the forward/backward substitution. Furthermore, the GEMM operation is known to be the best optimized operation on CPUs, as well as, on GPUs.
– In contrast to the LU decomposition, the GEMM operation inside the Gauss–Jordan-elimination scheme is constant in the number of affected rows and by choosing a proper block size the GEMM operation makes use of the whole device.
– By removing the data dependencies introduced by the forward/backward substitution scheme and only relying on the GEMM operation one can easily obtain and scalable distributed scheme to employ more than one GPU.

Finally, we compare two typical measures used for comparison of HPC systems, namely the flop rate and the energy efficiency in terms of GFlops/s/W [8]. Regarding Fig. 3 we observe that none of the MAGMA based approaches can compete at least with our single GPU code. Even if we have in mind that our approach needs 12.5 % more flops than the LU decomposition we still achieve a higher peak performance. Furthermore, it is easy to see that already small problems lead to a good utilization of the GPU in terms of the achieved flop rate. The two GPU implementations even obtains a good performance with moderate size problems and reaches a peak performance of 1.8 TFlops/s. Again one can see that the missing distributed triangular solve in MAGMA results in a stagnation of the performance. In contrast to the case where we only have one right hand side, in the linear system the forward/backward substitution needs more flops than the LU decomposition. Therefore, only having a distributed factorization is not enough to obtain a fast, and in

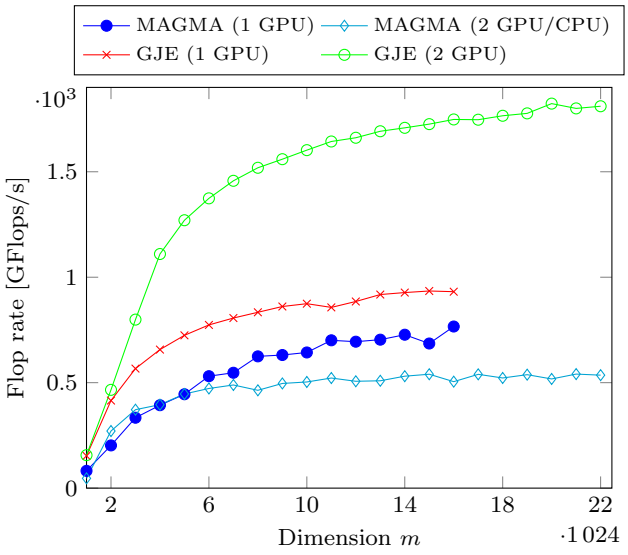


Fig. 3 Peak performance ($n = m$)

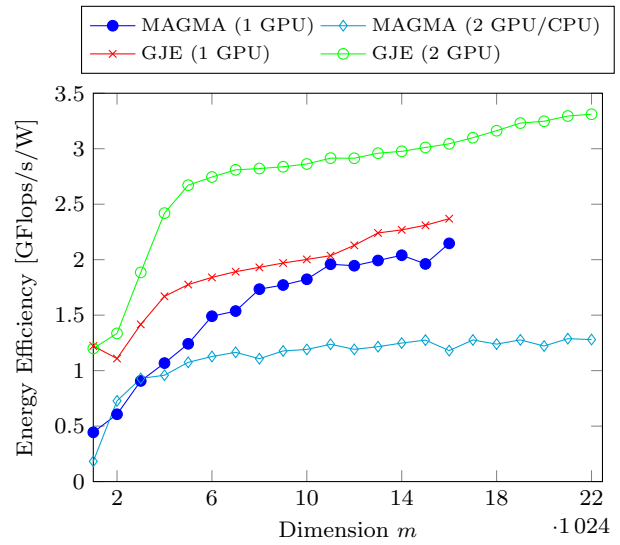


Fig. 4 Energy efficiency ($n = m$)

this way energy efficient, algorithm. Figure 4 shows that we get a value of 3.3 GFlops/s/W for our dual GPU implementation. Compared to the HPC systems in the current Green500 list (November 2015, [8]) we can compete with the Top-30 systems. Using this metric the maximum gain between the Gauss–Jordan-elimination method and the LU decomposition from MAGMA is between 1.5 and 2.5.

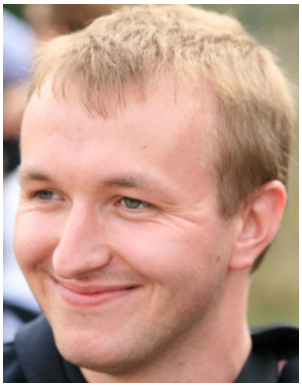
5 Conclusions

We showed that using the Gauss–Jordan-elimination scheme one can implement a fast and energy efficient solver for dense linear systems with many right hand sides. Furthermore, we showed that reducing memory transfers and data dependencies results in a scalable low-energy algorithm. By only requiring a high performance GEMM operation and minimal data dependencies on the accelerator devices this becomes a portable scheme for future architectures. Comparing the performance and the energy requirements to the MAGMA solution, we see that our implementation needs at most the same energy as the LU decomposition based solution process, but reduces the time to solution. Further optimizations like the influence of dynamic frequency scaling on the GPU and on the CPU are part of future research.

Acknowledgments Open access funding provided by Max Planck Institute for Dynamics of Complex Technical Systems.

References

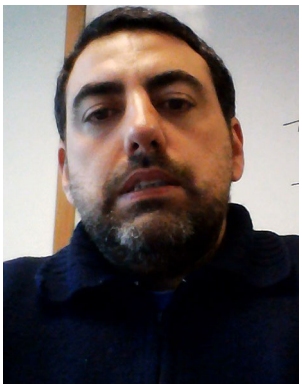
1. Benner P, Ezzatti P, Quintana-Ortí ES, Remón A (2013) Matrix inversion on CPU–GPU platforms with applications in control theory. Concur Comput Pract Exp 25(8):1170–1182. doi:10.1002/cpe.2933
2. Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S, Yamazaki I (2014) Accelerating numerical dense linear algebra calculations with GPUs. Numer Comput GPUs 3–28. doi:10.1007/978-3-319-06548-9_1
3. Ezzatti P, Köhler M (2016) Solving linear system with the augmented Gauss–Jordan-elimination on hybrid platforms. Tech. rep, Max Planck Institute for Dynamics of Complex Technical Systems
4. Freeh VW, Lowenthal DK, Pan F, Kappiah N, Springer R, Rountree BL, Femal ME (2007) Analyzing the energy-time trade-off in high-performance computing applications. IEEE Trans Parallel Distrib Syst 18(6):835–848. doi:10.1109/TPDS.2007.1026
5. Gardiner JD, Laub AJ (1986) A generalization of the matrix-sign-function solution for algebraic Riccati equations. Int J Control 44:823–832
6. Horowitz M, Indermaur T, Gonzalez R (1994) Low-power digital design. In: Proceedings of 1994 IEEE symposium on low power electronics, pp 8–11. doi:10.1109/LPE.1994.573184
7. Quintana-Ortí ES, Quintana-Ortí G, Sun X, van de Geijn R (2001) A note on parallel matrix inversion. SIAM J Sci Comput 22(5):1762–1771. doi:10.1137/S1064827598345679
8. Subramaniam B, Saunders W, Scogland T, Feng WC (2013) Trends in energy-efficient computing: a perspective from the Green500. In: 4th international green computing conference, Arlington, VA
9. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36(5–6):232–240. doi:10.1016/j.parco.2009.12.005
10. Tomov S, Nath R, Ltaief H, Dongarra J (2010) Dense linear algebra solvers for multicore with GPU accelerators. In: Proc. of the IEEE IPDPS'10, pp 1–8. IEEE Computer Society, Atlanta, GA. doi:10.1109/IPDPSW.2010.5470941



Martin Köhler received his Diploma in Mathematics from the TU Chemnitz in 2010. In 2011 Martin joined the Max Planck Institute for Dynamics of Complex Technical Systems in Magdeburg, where he maintains the HPC hardware of the Computational Methods in Systems and Control Theory group. His research focuses on high performance implementations of Numerical Linear Algebra algorithms with a special interest in efficient solvers Linear Systems and Matrix Equations.

Jens Saak received his Diploma in Industrial Mathematics from the University of Bremen in 2003. He then moved to the TU Chemnitz where he got his PhD in the Mathematics in Industry and Technology group in 2009. In 2010 Jens joined the Max Planck Institute for Dynamics of Complex Technical Systems in Magdeburg, where he has headed the Team on Scientific Computing in the Computational Methods in Systems and Control Theory department ever since. From 2013 on, he has additionally been Teamleader for Matrix Equations. His research interest span all aspects of the Linear Algebra of Matrix Equations, their applications in Control and Model Order Reduction and the scientific and high performance computing aspects of the underlying solvers.

Carolin Penke received her Bachelor's degree in Applied Mathematics from the Otto-von-Guericke-Universität Magdeburg in 2014. She is currently a Research Assistant at the Max Planck Institute for Dynamics of Complex Technical Systems in Magdeburg while pursuing her Master's degree. Her research focuses on Scientific Computing and the use of accelerators with applications in Numerical Linear Algebra.



Pablo Ezzatti received his bachelor and PhD degrees in Computer Sciences from the Universidad de la República (Uruguay) in 2001 and 2011. He is currently Associate Professor in the department of Computer Science of the Facultad de Ingeniería at the Universidad de la República. His research interests include scientific computing, parallel programming, linear algebra, as well as advanced architectures and hardware accelerators.