Balancing Energy and Performance in Dense Linear System Solvers for Hybrid ARM+GPU platforms

Juan P. Silva, Ernesto Dufrechou, Pablo Ezzatti
Facultad de Ingeniería, Universidad de la República,
11300, Montevideo, Uruguay,
{jpsilva,edufrechou,pezzatti}@fing.edu.uy,

and

Enrique S. Quintana-Ortí
Departamento de Ingeniería y Ciencia de Computadores,
Universidad Jaume I,
12.071, Castellón, Spain,
quintana@icc.uji.es

and

Alfredo Remón and Peter Benner
Max Planck Institute for Dynamics of Complex Technical Systems,
D-39106, Magdeburg, Germany,
{remon,benner}@mpi-magdeburg.mpg.de

Abstract

The high performance computing community has traditionally focused uniquely on the reduction of execution time, though in the last years, the optimization of energy consumption has become a main issue. A reduction of energy usage without a degradation of performance requires the adoption of energy-efficient hardware platforms accompanied by the development of energy-aware algorithms and computational kernels. The solution of linear systems is a key operation for many scientific and engineering problems. Its relevance has motivated an important amount of work, and consequently, it is possible to find high performance solvers for a wide variety of hardware platforms. In this work, we aim to develop a high performance and energy-efficient linear system solver. In particular, we develop two solvers for a low-power CPU-GPU platform, the NVIDIA Jetson TK1. These solvers implement the Gauss-Huard algorithm yielding an efficient usage of the target hardware as well as an efficient memory access. The experimental evaluation shows that the novel proposal reports important savings in both time and energy-consumption when compared with the state-of-the-art solvers of the platform.

Keywords: Dense Linear Systems, Gauss-Huard, NVIDIA Jetson K1, Energy-aware computing

1 Introduction

The solution of a system of linear equations of the form

$$Ax = b, \tag{1}$$

where $A \in \mathbb{R}^{n \times n}$ is a dense matrix, and $b, x \in \mathbb{R}^{n \times m}$ represent respectively, the right-hand side vector of independent terms (RHS) and the sought-after solution, is the main computational problem in the solution

of many scientific and engineering applications [1]. The traditional method to solve a general linear system is based on the LU factorization (i.e., Gaussian elimination), which must be complemented with (partial) row pivoting to ensure numerical stability [2]. The operation involves the factorization of the matrix A into the lower-triangular matrix $L \in \mathbb{R}^{n \times n}$ and the upper-triangular matrix $U \in \mathbb{R}^{n \times n}$ such that $A = LU$. Then, x is obtained via the solution of two triangular linear systems: $Ly = b$ and $Ux = y$. The factorization stage is the most time consuming operation of the process, and involves about $2n^3/3$ floating-point arithmetic operations (flops), in contrast with the n^2 flops of each subsequent triangular solver. The factorization phase and, therefore, the most time-consuming part of the method, can be expressed in terms of BLAS-3 operations [3]. Thus, an efficient implementation of this method on a parallel platform can potentially deliver remarkable performance. As a consequence, this is the method implemented in the LAPACK library (routine GESV), as well as in MATLAB® (command LINSOLVE or backslash).

The Gauss-Jordan elimination (GJE) is an efficient algorithm to compute the matrix inverse, and can be easily adapted to obtain the solution of linear systems of equations. The numerical stability can also be improved via partial pivoting. The main interest in this algorithm lies in its high degree of parallelism and its reduced number of memory operations. Consequently, this method usually delivers remarkable efficiency in modern hardware architectures that present a large number of computational units [4, 5]. However, when applied to the solution of linear systems, GJE incurs a larger computational cost than the traditional approach based on the LU factorization. In particular, it requires n^3 flops, compared with the $2n^3/3$ flops of the LU-based method. This limitation makes it useless as soon as the dimension of the system (n) becomes moderate. In the late 70s, P. Huard presented the Gauss-Huard algorithm (GH) [6], which can be considered as an extension of the GJE algorithm for the solution of linear systems, and represents an alternative and reliable method when complemented with column pivoting [7]. Interestingly, this method presents the same computational cost as the LU-based solver, i.e., $2n^3/3$ flops.

In recent years, hardware architectures have experienced remarkable changes. Mainly motivated by the limitations on power consumption, the strategy of increasing the processor frequency has been replaced by an increment in the number of computational units. GPUs and the Intel Xeon Phi processors are successful exponents of this trend. These new hardware platforms offer from tens to thousands of computational units, and are more energy-efficient than traditional multi-core processors. In this domain, the solutions presented by NVIDIA that integrate in a single board a low power ARM processor and a GPU occupy a relevant position. In particular, the most important example of this type of platforms is the NVIDIA Jetson TK1, composed of a quad-core ARM Cortex A15 processor and a Kepler GPU with 192 CUDA cores. However, to fully exploit the capabilities of these new platforms, the methods and their implementations must be reformulated.

This work aims to obtain a high-performance and energy-efficient linear system solver based on the Gauss-Huard method (with partial column pivoting) optimized for the NVIDIA Jetson TK1. We extend the work in [8] to further improve the new solvers and evaluate their performances in terms of runtime, but also energy. We include in the evaluation the state-of-the-art solvers in ARM and hybrid CPU-GPU platforms, specifically OpenBlas [9] and MAGMA [10]. Our experimental results show that one of the new implementations outperforms, in both runtime and energy consumption, the solvers included in the previous libraries, for problems of moderate to large dimensions ($n > 3,000$).

The rest of the paper is structured as follows: In Section 2, we describe three methods for the solution of linear systems based on the LU factorization, the GJE based strategy and the GH algorithm. Then, in Section 3, we present two new GPU-based versions that implement the GH algorithm, this is followed by an experimental evaluation of their performance and energy efficiency in Section 4. Finally, we discuss some conclusions and future work in Section 5.

2 Dense linear systems resolution

In this section, we revisit different approaches to compute the solution of a *dense* linear system $Ax = b$ with dense A. In numerical linear algebra (NLA), there are two main families of methods to solve this kind of problems: direct and iterative methods; see [2]. The first family consists of by deterministic methods that compute the exact solution of the problem (neglecting unavoidable floating-point rounding errors). On the other hand, iterative methods start with an initial solution and, after successive approximations, converge to a solution according to certain criteria. This usually means that a threshold, *tol*, is set such that when the difference between the computed solution and the exact solution is lower than *tol*, the method stops. From the computational point of view, direct methods generally present a computational cost of $O(n^3)$ flops and are mostly based on BLAS-3 operations, allowing an efficient use of the memory hierarchy in modern hardware platforms. In contrast, iterative methods present a computational cost of $O(n^2)$ flops per iteration and they do not take advantage of BLAS-3 operations which limits their performance. However, an iterative

method may be faster than a direct method if it presents a lower computational cost, namely, the number of iterations is considerably smaller than n (i.e., the convergence is fast or the problem does not require a high accuracy in the solution). On the other hand, direct methods allow to reuse most of the computations in the successive solution of linear systems with the same coefficient matrix. In this work, we focus on direct methods for the solution of dense linear systems. In more detail, in this section we describe three direct approaches based on the traditional LU, the GJE and the GH algorithms. All methods are in-place and, therefore, present a similar cost in terms of memory requirements.

2.1 Solution of linear systems via the LU factorization

The LU-based algorithm for the solution of linear systems is analogous to the traditional strategy based on Gaussian elimination. Specifically, the method consists of three steps that initially (first step) compute the LU factorization of A. This operation requires of a row pivoting strategy to ensure numerical stability [2]. The decomposition takes the form $PA = LU$, where $P \in \mathbb{R}^{n \times n}$ is a permutation matrix, and the factors $L, U \in \mathbb{R}^{n \times n}$ are, respectively, unit lower triangular and upper triangular. The L and U factors are stored in the memory space of A (i.e. in-place) to reduce the memory requirements. The main diagonal of L does not need to be stored since L is unit lower triangular. To further reduce memory use, P is implicitly stored as a vector. Then, the system is rewritten as $LUx = (Pb) = \hat{b}$ and, therefore, x can be obtained by solving the triangular linear systems $Ly = \hat{b}$ followed by $Ux = y$.

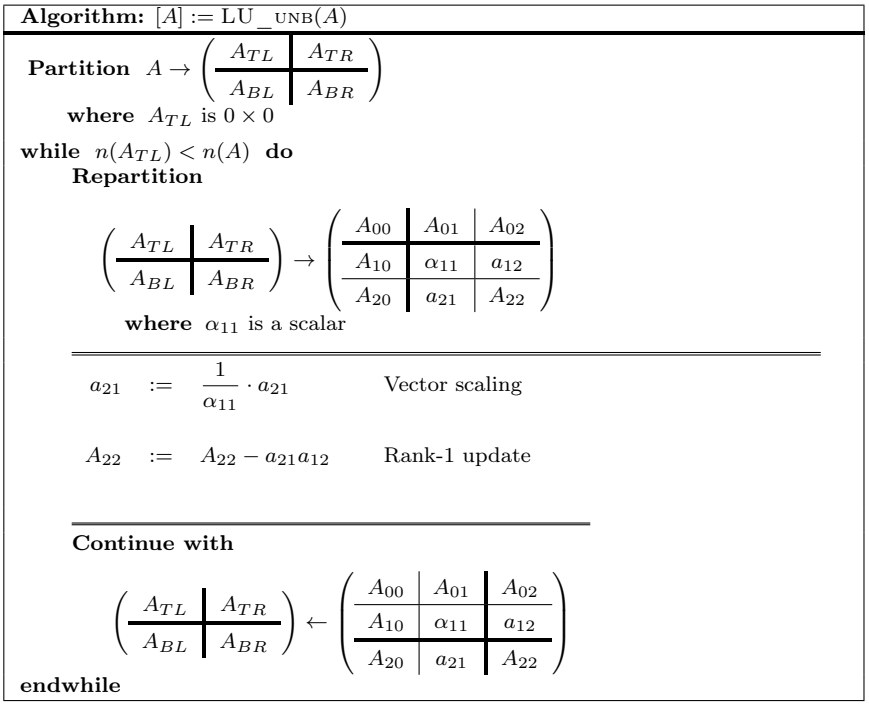

Figure 1: Scalar *in-place* algorithm to compute the LU factorization of matrix A. Upon completion, A is overwriten by the triangular factors L and U.

Most of the computational cost is due to the factorization of A. This can be exploited when several systems involving the same coefficient matrix must be solved. In this case, the factors L and U can be reused while the matrix decomposition needs to be computed only once. Consequently, the solution of the successive systems presents a computational cost of $2n^2$ flops. Figure 1 shows an in-place unblocked variant of the Gaussian elimination algorithm to compute the LU factorization using the FLAME notation [11, 12]. In the figure, $n(A)$ stands for the number of columns of A.

Figure 2 shows the blocked version of the method, where nb is the algorithmic block size. For simplicity, pivoting is not included in both algorithms. In general, blocked variants tend to offer better performance in modern hardware architectures, since they improve the computational intensity [2] and reduce the memory operations.

The LU-based method presents some drawbacks that limit its performance on massively parallel architectures. Concretely:

- The method consists of three sequential stages (LU factorization, upper triangular solve and lower

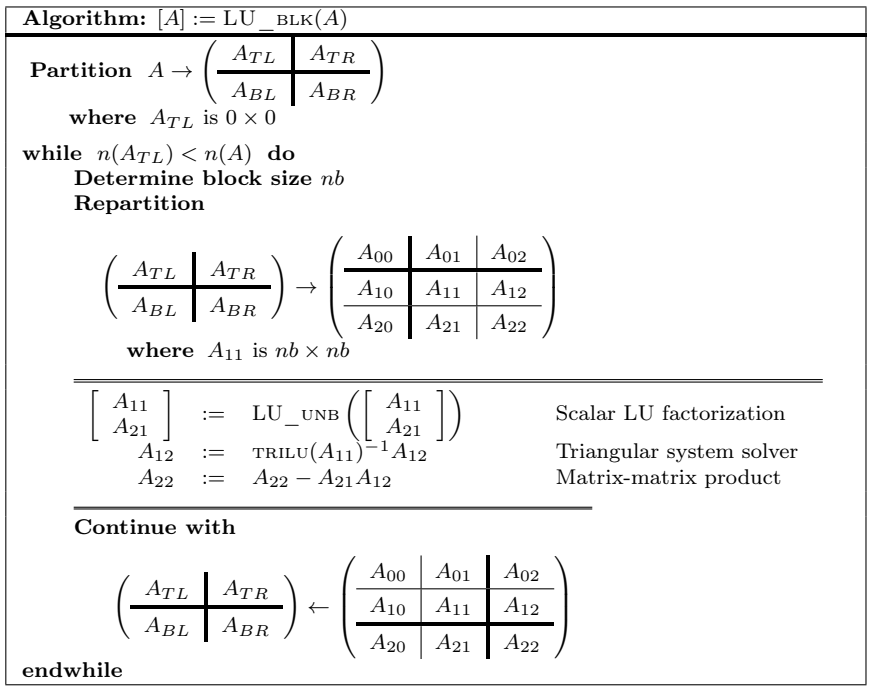

Figure 2: Blocked *in-place* algorithm to compute the LU factorization of A. Upon completion, A is overwriten by the triangular factors L and U.

triangular solve), implicitly defining two synchronization points.

- In addition, it requires the solution of two triangular linear systems, an operation that presents a rich collection of data dependencies and limited concurrency. Furthermore, small triangular linear systems also appear during the factorization stage.

These problems present little relevance when the dimension of A grows, since the factorization concentrates most of the computational effort. In this situation, the possibility to use BLAS-3 kernels during the factorization turns the method extremely suitable for parallel platforms.

LAPACK [13] is a specification that provides many important NLA operations, and offers high performance and numerically reliable implementations for many types of platforms. Certain implementations may include a number of high performance computing (HPC) techniques and also particular customizations for different processors and architectures. The LAPACK specification includes support for the different stages of the LU-based solver. In particular, the routine GETRF computes the LU factorization (applying partial row pivoting) of a non-singular matrix, while the routine GETRS allows the solution of the subsequent triangular systems from the factors computed by GETRF. Additionally, LAPACK includes the GESV driver routine for the solution of linear systems, which simply performs the appropriate calls to the routines GETRF and GETRS.

2.2 The Gauss-Jordan method

An alternative method for the solution of dense linear systems is the so-called Gauss-Jordan Elimination method (GJE). This algorithm calculates the inverse of a matrix, but can be easily adapted to obtain the solution of a linear system. Additionally, if row pivoting is employed, the algorithm exhibits stability properties similar to those of the Gaussian Elimination method [14].

The solution of linear systems with this method implies the application of GJE to the extended matrix $\hat{A} = [A, b]$. The method processes the extended matrix from left to right, updating, at each iteration, all the entries of $\hat{A}$. For the solution of linear systems, only the elements to the right of the active column need be updated, reducing the computational cost to n^3 flops. Once the process is completed, the solution of the linear system (i.e., the x vector) is stored in the last column of the extended matrix $\hat{A}$.

Figure 3 shows the blocked version of the GJE to solve the linear system of equations associated with A. For simplicity we do not include pivoting in the description. A description of a scalar version of the GJE method, that is invoked in the blocked version, can be found in [15]. It should be noted that this is also an *in-place* method, which means that, when the process is completed, the matrix A is overwritten with the transformed matrix and x overwrites b (in the last column of $\hat{A}$).

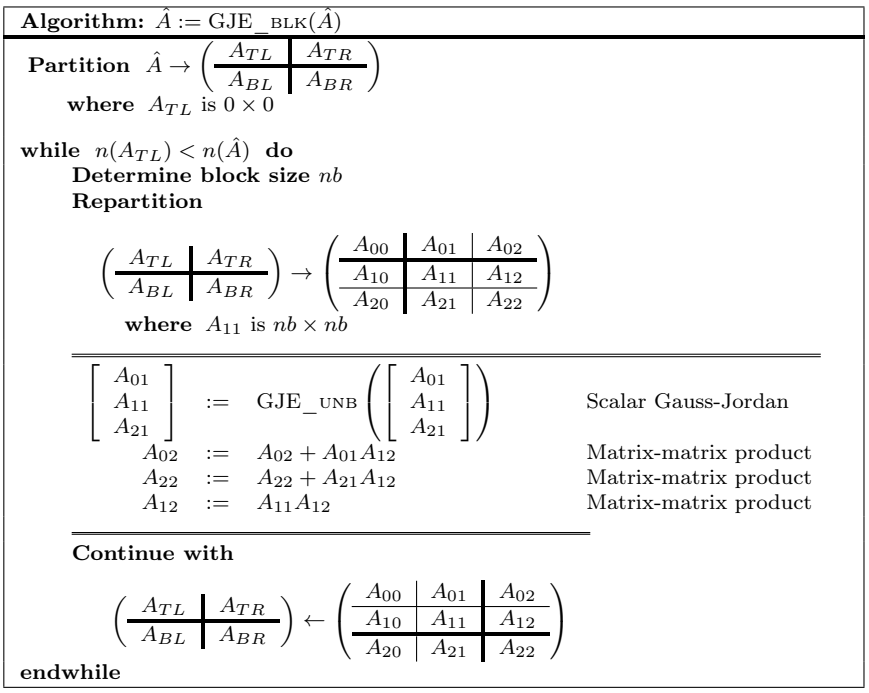

Figure 3: Blocked GJE algorithm for the solution of linear systems of the form $Ax = b$. Initially, $\hat{A} = [A, b]$. Upon completion, the solution x overwrites the last column of $\hat{A}$.

This method is rich in BLAS-3 operations. In particular, the updates $\hat{A}$ are performed via matrix-matrix products. This usually yields a high throughput when executed on massively parallel platforms. On the other hand, the use of the GJE method to solve a linear system requires n^3 flops, in contrast to the $2/3n^3$ flops for the LU-based method. In consequence, this method cannot compete with the LU-based algorithm for the solution of systems of large dimension.

2.3 The Gauss-Huard method

As previously stated, the GJE method presents some features that turn it especially appealing for its implementation on massively parallel platforms. Unfortunately the algorithm also presents a high computational cost for the solution of linear systems of equations compared with the LU-based method. This drawback was addressed by P. Huard [6] in the late 1970s. In his work, he presents a smart variant of the GJE algorithm to solve linear systems that incurs a computational cost analogous to that of the LU-based method. This method, known as the Gauss-Huard (GH) algorithm, presents a convenient memory access pattern. But this feature is destroyed if row pivoting is introduced to ensure numerical stability. This problem was overcome by T. J. Dekker et al. in [16, 7]. They proposed the inclusion of column pivoting and proved that the resulting method offers numerical characteristics similar to those of the LU factorization with row pivoting.

Figure 4 describes the GH for the solution of a linear system of equations. Pivoting is not included in the algorithm for simplicity, although all our variants integrate partial column pivoting. In practice, pivoting can be easily added as follows: before the diagonalization of $\left[\hat{\alpha}_{11},\ \hat{a}_{12}^T\right]$, this vector is searched for its maximum entry in magnitude (excluding its last element, which corresponds to an entry of b) and the column of $\hat{A}$ corresponding to this entry is then swapped with the column of $\hat{A}$ containing the diagonal entry $\hat{\alpha}_{11}$.

A blocked variant of GH was introduced for distributed-memory (message-passing) systems in [17]. Unfortunately, the authors did not perform any experimental evaluation of the implementation, and simply stated that its performance could be expected to be close to that of an LU-based solver.

Figure 5 describes a blocked version of the Gauss-Huard algorithm which processes nb columns of the matrix per iteration.

3 Gauss-Huard implementations for hybrid CPU-GPU platforms

CPU-GPU platforms have currently reached a prominent position in HPC, as revealed by the top positions of the Top500 list [18]. They are also widely used in the domain of numerical methods and linear algebra. This is reflected by the efforts integrated in modern scientific numerical libraries to exploit this kind of platforms;

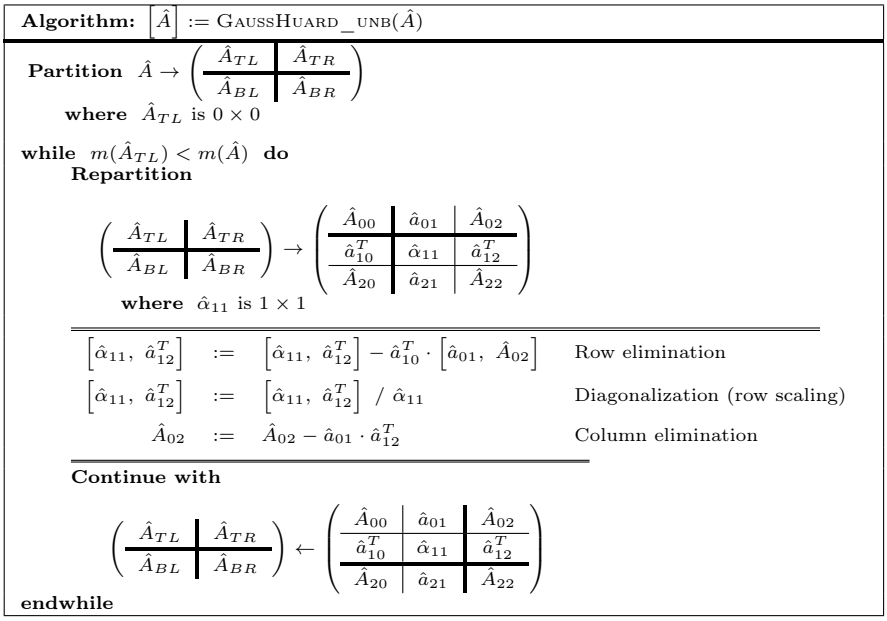

Figure 4: Scalar (unblocked) Gauss-Huard algorithm for the solution of linear systems of the form $Ax = b$. Initially, $\hat{A} = [A, b]$. Upon completion, the solution x overwrites the last column of $\hat{A}$.

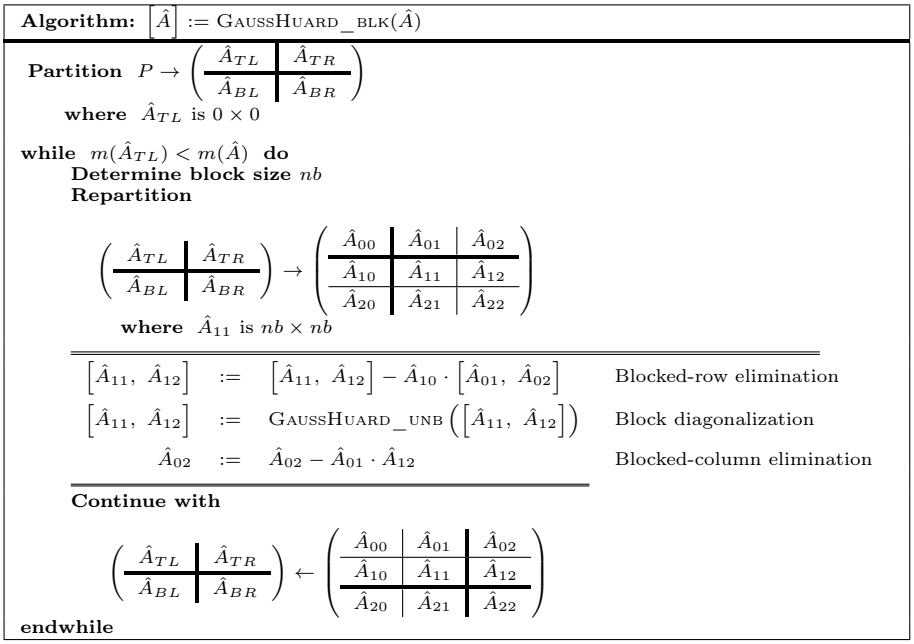

Figure 5: Blocked Gauss-Huard algorithm for the solution of linear systems of the form $Ax = b$. Initially, $\hat{A} = [A, b]$. Upon completion, the solution x overwrites the last column of $\hat{A}$.

see e.g., MAGMA [19], PETSc [20] or ViennaCL [21]. All those libraries include routines to solve (dense and sparse) linear systems taking advantage of the capabilities provided by this kind of hybrid architectures. Current GPUs offer not only an ample parallelism, but also remarkable flops-per-Watt ratios and competitive economical cost. This has motivated the development of boards that embed one or more general-purpose low-power processors (like those developed for mobile devices by ARM) with low-power GPUs. The performance and moderate power consumption of such hybrid boards has motivated the assembly of powerful clusters with hundreds of these boards. This is the case of the Tibidabo supercomputer at the Barcelona Supercomputer Center [22].

The GH algorithm combines highly parallel operations suitable for GPUs and fine-grain parallelism computations (mainly due to pivoting) that suits general-purpose processors. Thus, this algorithm represents a good candidate for hybrid CPU-GPU platforms.

In this context, we present two implementations of the GH algorithm that target hybrid CPU-GPU low-power platforms. The first variant performs all computations in the GPU. The second variant distributes the computations between the CPU and the GPU, but incurs in a overhead due to data transfers between the CPU and the GPU memories.

3.1 GPU version (GH_{gpu})

Due to the moderate computational power offered by the general-purpose processor when compared with that offered by the GPU, this implementation performs all the computations in the latter. On the other hand, the main drawback of GH_{gpu} is that not all the computational units in the platform are employed. This variant is an implementation of algorithm GAUSSHUARD_BLK using the kernels offered by the CUBLAS library, i.e., the implementation of the BLAS provided by NVIDIA.

In an initial phase, the whole $\hat{A}$ matrix is transferred from the CPU to the GPU. Then, the algorithm proceeds in the GPU, and upon completion, the last column of $\hat{A}$, containing x, is transferred back to the CPU memory.

As previously stated, most of the computational effort in algorithm GAUSSHUARD_BLK lies in the matrix-matrix products. To fully exploit the capabilities of the GPU during the execution of these products, the matrices involved must be of a moderate to large size. The dimension of these matrices is determined by the system dimension, n, and the algorithmic block size, nb. Consequently, a small value of nb limits the performance of this variant. On the other hand, a large value of nb increases the relative computational effort of the *block diagonalization*, which is a BLAS-2 operation that delivers limited performance. To partially overcome this problem, we included a multiple nested algorithmic block sizes strategy in GH_{gpu}. Specifically, the block diagonalization is performed via a blocked variant of the GH algorithm, i.e., GAUSS-HUARD_BLK. As a result, BLAS-3 kernels can be employed during most of this computation. To maximize the performance of GH_{gpu}, both algorithmic block-sizes must be carefully chosen.

3.2 Hybrid Version(GH_{hyb})

This variant aims to exploit both available computational resources in the platform, i.e., the CPU and GPU. To achieve this, it requires data transfers between the CPU and the GPU and, consequently, incurs a certain overhead due to communications. To alleviate this, as an initial phase, the matrix $\hat{A}$ is transferred from the CPU memory to the GPU memory. With this we avoid to transfer the panel $[\hat{A}_{11}, \hat{A}_{12}]$ from CPU to GPU at each iteration of the algorithm. The total amount of data transferred remains the same but, given the platform specifications, a single larger data transfer is more efficient than several smaller transfers. Furthermore, this data transfer can be partly overlapped with the diagonalization of the first panel. This way, we save time as well as energy.

Most of the computations in the algorithm are cast in terms of matrix-matrix products. This operation exhibits a high level of concurrency and suits the GPU architecture. In contrast, the block diagonalization presents a moderate cost and its concurrency is constrained by the pivoting scheme. Thus, the natural way to distribute the operations is to perform the matrix-matrix products in the GPU and the block diagonalization in the CPU. Therefore, the bulk of the computations are performed in the most powerful processor, the GPU. This partitioning of the workload (diagonalization in the CPU and the remaining block eliminations in the GPU) requires that, at each step of the GAUSSHUARD_BLK algorithm, the block $[\hat{A}_{11},\ \hat{A}_{12}]$ is sent from the GPU (after completing the *block-row elimination*) to the CPU and retrieved from there back into the GPU (after the computation of the block diagonalization). Consequently, the volume of data transferred at each iteration of the algorithm is proportional to n and nb. After the procedure is completed, the solution vector (stored in the last column of $\hat{A}$) must be sent to the CPU.

Similarly to the fully GPU implementation, GH_{gpu}, the efficiency of GH_{hyb} strongly depends on the value of nb: small values of nb close to 32, 64, limit the performance of the matrix-matrix products performed in the GPU, though this ensures that the execution of the algorithm is driven by computation instead of data transfers. However, a large value of nb may hurt the performance of GH_{hyb} because a significant part of the computations is mapped to the less powerful processor, the CPU. Again, we propose to perform the block diagonalization via a blocked variant of the algorithm and, therefore, we leverage a double algorithmic block size, namely nbd and nbg. The former is uniquely employed for the computations in the ARM processor, i.e., during the block diagonalization. This way, BLAS-3 kernels can be employed during this stage, accelerating its execution. At a higher level, in GAUSSHUARD_BLK we set $nbg = nb$. A more efficient execution of the block diagonalization phase permits to chose larger values for nbg, which also yield higher performance during the execution of the matrix-matrix products in the GPU. Additionally, larger values of nbg reduce the number of iterations and, thus, the number of data transfers (despite it might increase the amount of data transferred, transfers will be performed in a more efficient manner).

Additionally, GH_{hyb} makes a heavy use of tuned computational kernels in OpenBLAS and NVIDIA CUBLAS, respectively, for the ARM and the GPU processors. Moreover, a few key scalar and Level-1 BLAS operations are performed via *ad-hoc* kernels, parallelized with OpenMP, which yield higher performance than their counterparts in OpenBLAS.

4 Experimental evaluation

In this section we show the experimental evaluation of the GH-based solvers introduced in the previous section. The evaluation focuses on two aspects, performance (measured in execution time and GFLOPS) and energy consumption. All experiments are performed employing single precision arithmetic. We compare the solvers on the solution of randomly generated systems of dimension $n = 1,024,\ldots,7,168$.

The novel codes has been tuned for the hardware platform employed, and compared with two LU-based solvers considered as the state-of-the-art for ARM processors and hybrid CPU-GPU platforms. In particular, the solver in OpenBLAS v.0.2.14 to exploit the ARM processor and its counterpart in MAGMA v.1.6.2 to leverage the CPU and GPU processors. Note that the routine in MAGMA internally uses kernels in OpenBLAS to perform computations in the ARM.

4.1 Experimental platform

The experimental hardware platform, a JETSON TK1 [23] board, is composed of a Tegra K1 SoC (Systems on a Chip) processor with 2GB of DDR3L RAM. In more detail, the Tegra K1 includes an NVIDIA GPU with 192 CUDA Kepler cores and an ARM quad-core Cortex-A15 processor at 2.32 GHz.

The board runs an extended version of Linux Ubuntu 14.04 adapted for the ARM architecture. The codes for the ARM are compiled using *gcc* v.4.8.2, while the codes developed for the NVIDIA GPU are compiled with *nvcc* v.6.0.1.

The platform configuration was adapted such that the maximum performance level was reported with a negligible loss of energy efficiency. This involved to: (1) set the frequency governor of the CPU to `performance` (instead of its default value, `ondemand`), (2) turn on the ARM-cores as required, (3) and set the GPU clock to the highest available frequency, i.e., $852,000,000$ `Hz` (by default set to $12,000,000$ `Hz`).

4.2 Performance evaluation

The runtimes showed for the different GPU-variants include the overhead associated with data transfers between CPU and GPU memory spaces.

Table 1: Execution time (in seconds) and performance (in GFLOPS) attained with the solvers.

	MAGMA				OpenBLAS			
	1-THREAD		4-THREADS		1-THREAD		4-THREADS	
n	GFLOPS	TIME	GFLOPS	TIME	GFLOPS	TIME	GFLOPS	TIME
1K	0.40	1.81	0.44	1.62	3.81	0.19	11.18	0.06
2K	1.09	5.24	1.18	4.87	4.07	1.41	13.57	0.42
3K	1.80	10.74	1.91	10.14	4.20	4.61	14.69	1.32
4K	2.38	19.26	2.61	17.58	4.10	11.17	14.47	3.17
5K	3.09	28.98	3.36	26.65	4.29	20.88	15.62	5.73
6K	3.76	41.17	4.05	38.18	4.29	36.04	15.74	9.82
7K	4.41	55.74	4.74	51.75	4.32	56.77	15.97	15.38
	GH_{hyb}				GH_{gpu}			
	1-THREAD		4-THREADS		1-THREAD		4-THREADS	
n	GFLOPS	TIME	GFLOPS	TIME	GFLOPS	TIME	GFLOPS	TIME
1K	2.44	0.29	2.14	0.33	1.50	0.48	-	-
2K	2.65	2.16	2.14	2.67	5.29	1.08	-	-
3K	2.95	6.55	2.68	7.21	10.53	1.84	-	-
4K	3.14	14.59	3.08	14.85	15.06	3.04	-	-
5K	3.28	27.32	3.59	24.96	18.59	4.81	-	-
6K	3.40	45.52	4.06	38.09	21.95	7.04	-	-
7K	3.50	70.24	4.25	57.80	24.56	9.99	-	-

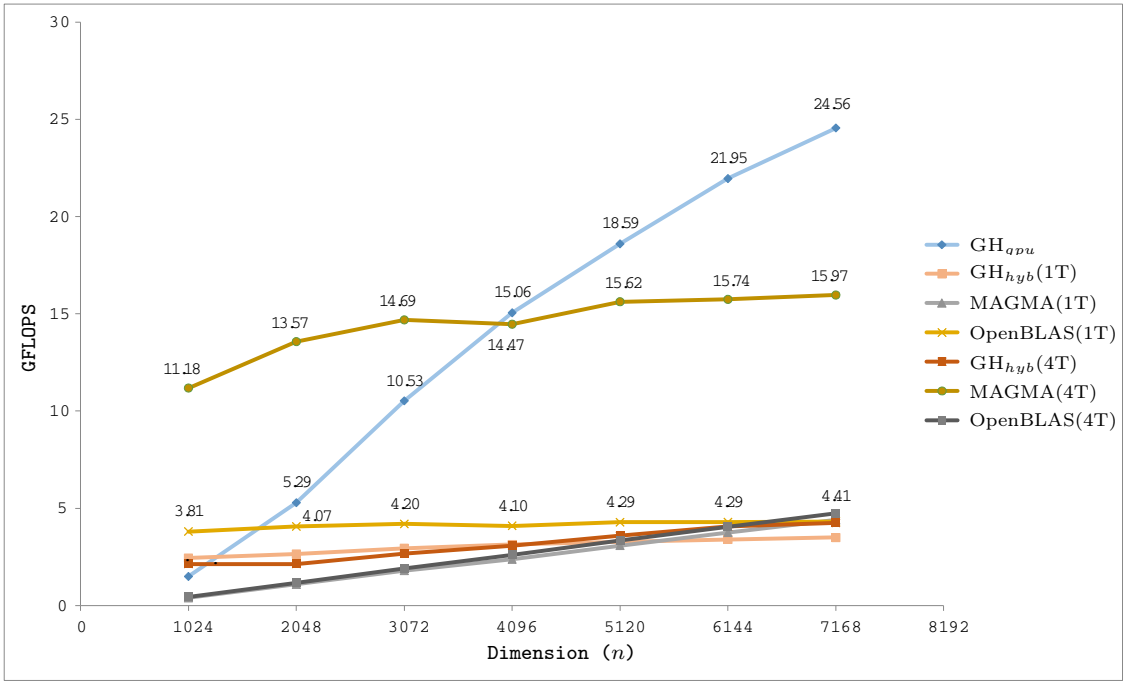

Figure 6: Power-efficiency (measured in GFLOPS/Watt) attained by the OpenBLAS and GH$_{gpu}$ solvers.

The quality of the numerical results obtained with all solvers are numerically equivalent. In other words, the residual error associated with all the computed solutions are of the same order.

Table 1 summarizes the runtime (in seconds) and performance (in GFLOPS or billions of flops per second) obtained by the four solvers, i.e., OpenBLAS, MAGMA, GH$_{gpu}$ and GH$_{hyb}$. Additionally, for each variant (except GH$_{gpu}$), we include the number of threads employed on the CPU. In particular, we present two options for each variant, corresponding to the use a single thread and use 4 threads (one per core in the Cortex A-15 processor). For GH$_{gpu}$ and GH$_{hyb}$, we evaluate several block sizes (both, internal and external), but for simplicity, only the best performance for each system dimension is shown. A graphical comparison of the performance attained by all variants is offered in Figure 6.

First of all, we focus on the results obtained with solvers in OpenBLAS and MAGMA. The solver in the MAGMA library reports a moderate performance on the solution of small to moderate-scale systems. This might be caused by the CPU-GPU communication overhead. The performance rapidly improves with the problem dimension. Additionally, the use of 4 threads (note that MAGMA offers a hybrid CPU-GPU method) delivers marginal gains, about a 10% of time reduction. OpenBLAS offers a remarkable performance despite it uses only the ARM processor (that presents a lower computational power than the GPU), but its performance is far from that of GPU-based solvers. ts parallel version is able to beat the MAGMA solver for all problem dimensions.

Regarding the GH-based solvers, the hybrid variant (GH$_{hyb}$) offers execution times comparable with those obtained with the MAGMA library for the larger problems, but it is clearly superior on the solution of small systems, especially when a single thread is employed in the CPU. In contrast, the GH$_{gpu}$ variant, which performs all the computations in the GPU, achieves a remarkable performance, especially for larger problems. This is because the capabilities of the massively parallel architecture in the GPU can be exploited more efficiently in the solution of large systems. It should be note that GH$_{gpu}$ is the best option for problems of dimension n larger than $3,072$.

In a conclusion, the hybrid implementations do not attain high performance because of the high overhead introduced by the data transfers. For the solution of small problems, the use of the ARM processor only is convenient since it does not require of data transfer. For larger problems, the higher computational power of the GPU promotes the GPU-only variant as the best option.

4.3 Power consumption evaluation

In this section we address the power and energy evaluation of the two more efficient solvers, i.e., the solver in the OpenBLAS library and GH$_{gpu}$. We exclude from this comparison the hybrid solvers (GH$_{hyb}$ and the solver in the MAGMA library) because the power required by them is expected to be higher than that of the solvers that are fully executed in one of the processors in the platform. Additionally, they exhibited a higher

runtime and thus, larger energy requirements are to be expected by the hybrid solvers. For the OpenBLAS solver we evaluate its behavior with 1 and 4 threads.

In order to measure the power consumption, we connected a WattsUp?Pro wattmeter (accuracy of ±1.5% and a rate of 1 sample/sec.) to the power line from the electric socket to the power supply unit (PSU) of the Tegra TK1 device. Thus we measure the power consumption of the whole board. The power measurements are collected, stored, and treated in a different server, so that the measurement does not disturb the solvers execution. All tests were executed for a minimum of 1 minute, after a warm-up period of 2 minutes.

Table 2 collects the average power (P_{avg}), total energy (E_{tot}), and GFLOPS/Watt reported by the solvers. Figure 7 shows the GFLOPS/Watt ratio of the solvers. The results show that, when the system dimension $n \leq 2,000$, the most energy-efficient solver is that provided by the OpenBLAS library (using 4 threads). This is explained by the absence of expensive CPU-GPU data transfers. Although the GPU architecture is more energy-efficient than the ARM processors, it cannot be fully exploited for the solution of small problems. Thus, the time and energy spent in data transfers is superior to the gains reported by the GPU usage. This is exposed by the P_{avg} values. GH_{gpu} delivers a lower P_{avg} but a longer execution time (see Table 4.2), resulting in higher energy consumption. For larger systems, the GH_{gpu} reported the best results in terms of energy. For this kind of problems, the full exploitation of the GPU capabilities compensates the time and energy spent in data transfers (note than only two data transfers are required in GH_{gpu}, initially the whole $\hat{A}$ matrix is sent to the GPU and, upon completion, the x vector is retrieved by the CPU). A special case occurs when $n = 3,000$. In this case, the solver in OpenBLAS (with 4 threads) reports the shorter execution time, but due to the lower P_{avg} of GH_{gpu}, the latter consumes less energy. If we focus on the P_{avg} results, the OpenBLAS solver with a single thread presents the lowest power consumption. This is explained by the fact that the solver uses a single core of the ARM processor. On the other hand, when the same solver operates with 4 threads, it delivers a speed-up of about 3.5, while the power is nearly doubled. Consequently, the use of 4 threads in OpenBLAS is more energy-efficient. The average power consumed by GH_{gpu} (remember this solver is completely executed in the GPU) is lower than that of the parallel OpenBLAS kernel (using 4 cores of the ARM processor), reporting up to 50% of power reduction depending on the system dimension. This implies that the GPU presents a higher peak performance and also a lower energy-consumption and thus, this architecture specially appealing for our purposes.

Table 2: Average power consumption (in Watts), total energy (in Joules) and performance per Watt (in GFLOPS/Watt) obtained for the three solvers on the solution of linear systems of equations of dimension n.

	OpenBLAS						GH_{gpu}		
	1-thread			4-threads			1-thread		
n	P_{avg}	E_{tot}	$GFLOPS/Watt$	P_{avg}	E_{tot}	$GFLOPS/Watt$	P_{avg}	E_{tot}	$GFLOPS/Watt$
1K	6.3	1.18	0.60	12.6	0.80	0.89	6.1	2.92	0.25
2K	6.3	8.86	0.65	13.2	5.57	1.03	6.9	7.47	0.77
3K	6.3	29.02	0.67	13.7	17.99	1.07	8.1	14.87	1.30
4K	6.3	70.36	0.68	13.7	43.26	1.06	9.2	27.99	1.64
5K	6.3	131.51	0.68	14.0	80.31	1.11	9.9	47.64	1.88
6K	6.3	227.07	0.68	14.1	138.77	1.11	10.7	75.36	2.05
7K	6.3	357.66	0.69	14.2	218.37	1.12	10.9	108.99	2.25

Since the platform contains several devices —e.g., network interface cards— we measured the average power while idle for 1 minute, P_I, showing a value of 2.6 Watts. This correspond to the power consumption of the board when no operation is executed. We use this value to compute the *net energy*, corresponding to the energy consumption that is obtained after subtracting P_I from the power samples. The *net energy* approximates the actual energy spent in computations, and allows a fair comparison between the computational kernels. Figure 8 shows the GFLOPS/Watt computed with the average net power (left) and the *net energy* of the solvers. Both figures show the superiority of GH_{gpu}, because it performs more arithmetic operations per Watt and also requires less energy.

5 Concluding Remarks

During the last years, energy consumption has become a major issue in high performance computing, too. In this work we present two efficient solvers for linear sytems of equations that present remarkably low energy consumption. The new solvers exploit the capabilities of a low-power hardware platform, an NVIDIA

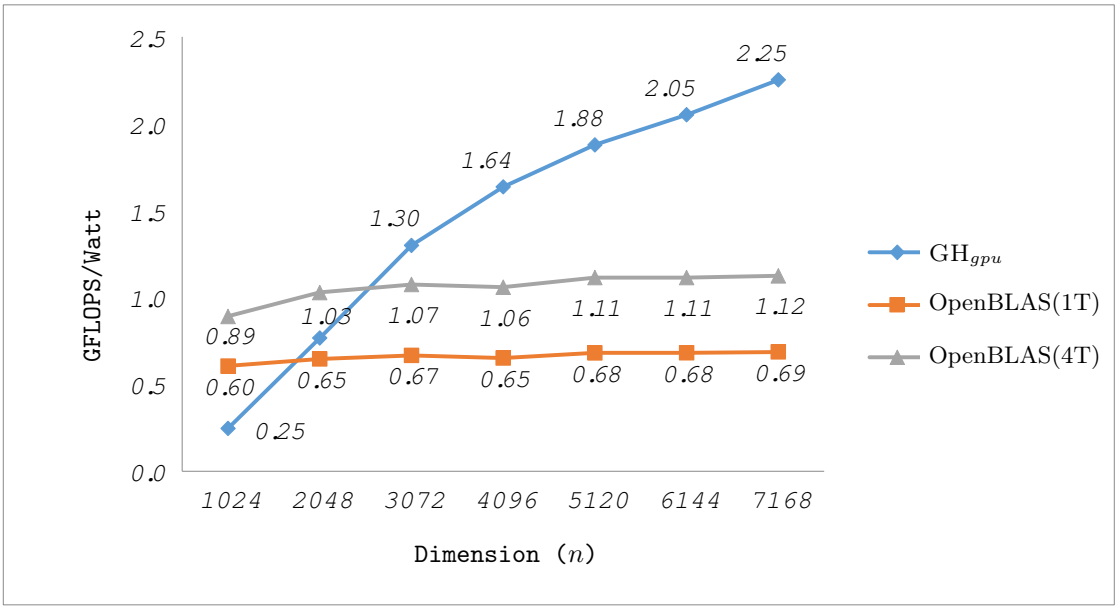

Figure 7: Energy-efficiency (measured in GFLOPS/Watt) attained with the OpenBLAS and GH$_{gpu}$ solvers.

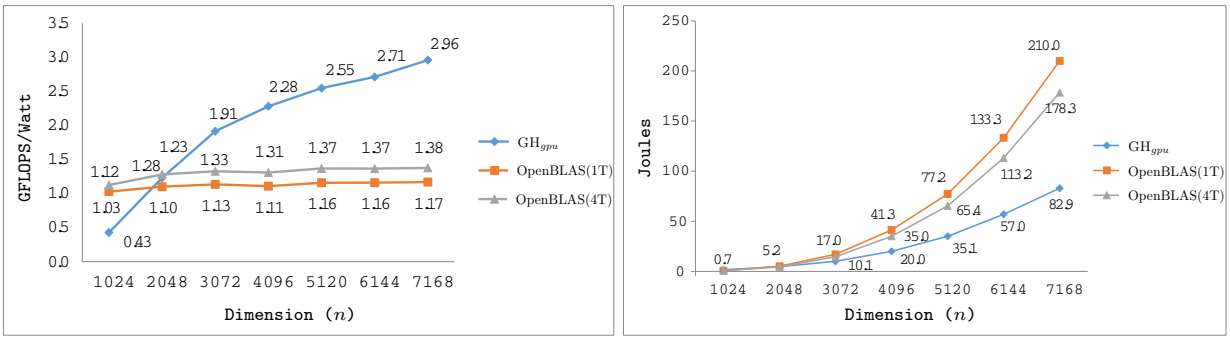

Figure 8: Net energy-efficiency measured in GFLOPS/Watt (left) and total net energy in Joules (right), attained with the OpenBLAS and GH$_{gpu}$ solvers.

JETSON K1 (formed by an ARM quad-core processor and a NVIDIA GPU with 192 CUDA cores) device to efficiently run a Gauss-Huard-based solver. The Gauss-Huard algorithm presents some features which turn it more suitable for the target hardware platform than the traditional LU-based solver. The GH$_{hyb}$ solver aims to exploit all the computational units in the platform concurrently. Despite it is able to perform computations concurrently in both processors, it also requires data transfers between their memory spaces, incurring into a considerable overhead. The GH$_{gpu}$ solver performs all the computations in the most powerful processor, the GPU. Thus, it incurs a minor communication overhead that is compensated with both higher performance and energy-efficiency of the GPU. The performance of both solvers is compared with the solvers in OpenBLAS and MAGMA libraries. The experimental results show the superior behavior of the GH$_{gpu}$ solver on the solution of linear systems of dimension $n \geq 4,000$. For smaller systems, the OpenBLAS solver reports the best execution times because it runs entirely in the ARM processor and does not require any CPU-GPU communication. The power and energy evaluation show the convenience of the GPU processor over the ARM quad-core. The GPU consumes less power and delivers a higher GFLOPS rate. Consequently, the GH$_{gpu}$ variant attains the best energy-efficiency on the solution of moderate to large-scale systems. On the solution of small systems, again the absence of CPU-GPU communications in the OpenBLAS kernel turns it into the most energy-efficient variant. The communication overhead affects negatively the performance (and therefore the energy efficiency) of the hybrid solvers (i.e., the solver in the MAGMA library and GH$_{hyb}$) and thus, both deliver limited performance.

There are different aspects that were not addressed in the current work but deserve more investigation. For example, it is important to study new techniques to improve the results of the hybrid solvers. For example techniques like *look-ahead* [24] have demonstrated important benefits in similar applications. It

should also be evaluated how the hybrid CPU-GPU implementations can benefit from the improvements in the Unified-Memory-Access announced by NVIDIA for their future devices. Additionally, we plan to extend this study to other low power consumption hardware platforms, e.g., Samsung ODROID-XU4. Finally, although this work targets single precision arithmetic (due to the low performance of the architecture on double precision), we plan to investigate how GH can be complemented with an iterative refinement to attain a full double precision solution.

Acknowledgment

Juan Silva, Ernesto Dufrechou and Pablo Ezzatti acknowledge the support from Programa de Desarrollo de las Ciencias Básicas, and Agencia Nacional de Investigación e Innovación, Uruguay. Enrique S. Quintana-Ortí was supported by project TIN-2014-53495-R of the *Ministerio de Economía y Competitividad* and FEDER. All researchers acknowledge the support from the EHFARS project funded by the German Ministry of Education and Research BMBF.

References

[1] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[2] G. Golub and C. V. Loan, *Matrix Computations*, 3rd ed. Baltimore: The Johns Hopkins University Press, 1996.

[3] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, no. 1, pp. 1–17, March 1990.

[4] P. Ezzatti, E. Quintana-Ortí, and A. Remón, "Using graphics processors to accelerate the computation of the matrix inverse," *The Journal of Supercomputing*, vol. 58, pp. 429–437, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11227-011-0606-4

[5] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, and A. Remón, "Matrix inversion on CPU-GPU platforms with applications in control theory," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1170–1182, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.2933

[6] P. Huard, "La méthode simplex sans inverse explicite," *EDB Bull, Direction Études Rech. Sér. C Math. Inform. 2*, pp. 79–98, 1979.

[7] T. J. Dekker, W. Hoffmann, and K. Potma, "Stability of the Gauss-Huard algorithm with partial pivoting," *Computing*, vol. 58, pp. 225–244, 1997.

[8] J. P. Silva, E. Dufrechou, E. Quintana-Ortí, A. Remón, and P. Benner, "Solving dense linear systems with hybrid CPU-GPU platforms," in *Proceedings of the XLI Latin American Computing Conference, CLEI 2015, Arequipa, Peru, 2015.* IEEE, to appear.

[9] OpenBLAS website, Z. Xianyi; [accessed 2015 Dec], http://www.openblas.net/.

[10] B. J. Smith, "R package MAGMA: Matrix algebra on GPU and multicore architectures, version 0.2.2," September 3, 2010, [On-line] http://cran.r-project.org/package=magma.

[11] 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.*, vol. 27, no. 4, pp. 422–455, 2001.

[12] 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.*, vol. 31, no. 1, pp. 1–26, 2005.

[13] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia: SIAM, 1992.

[14] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[15] E. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn, "A note on parallel matrix inversion," *SIAM J. Sci. Comput.*, vol. 22, pp. 1762–1771, 2001.

[16] T. J. Dekker, W. Hoffmann, and K. Potma, "Parallel algorithms for solving large linear systems," *Journal of Computational and Applied Mathematics*, vol. 50, no. 1–3, pp. 221–232, 1994.

[17] W. Hoffmann, K. Potma, and G. Pronk, "Solving dense linear systems by Gauss-Huard's method on a distributed memory system," *Future Generation Computer Systems*, vol. 10, no. 2–3, pp. 321–325, 1994.

[18] TOP500.org; [accessed 2015 Dec], http://www.top500.org/.

[19] MAGMA, Univ. of Tennessee; [accessed 2015 Dec], http://icl.cs.utk.edu/magma/.

[20] S. Balay, W. Gropp, L. C. McInnes, and B. Smith, "PETSc 2.0 users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11, October 1996.

[21] TU Wien and FASTMathSciDAC Institute, http://viennacl.sourceforge.net/.

[22] Barcelona Supercomputing Center, http://www.bsc.es.

[23] NVIDIA, http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html.

[24] P. Strazdins, "A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization," Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, Tech. Rep. TR-CS-98-07, 1998.