Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

GPU-based parallel algorithms for sparse nonlinear systems[☆]

V. Galiano [a], H. Migallón [a], V. Migallón [b,*], J. Penadés [b]

[a] Department of Physics and Computer Architectures, University Miguel Hernández, E-03202, Elche, Alicante, Spain
[b] Department of Computer Science and Artificial Intelligence, University of Alicante, E-03071, Alicante, Spain

ARTICLE INFO

Article history:
Received 1 March 2011
Received in revised form
5 September 2011
Accepted 28 October 2011
Available online xxxx

Keywords:
GPGPU
GPU libraries
Multicore architectures
Nonlinear conjugate gradient algorithms
Parallel preconditioners
Bratu problem

ABSTRACT

In this work we describe some parallel algorithms for solving nonlinear systems using CUDA (Compute Unified Device Architecture) over a GPU (Graphics Processing Unit). The proposed algorithms are based on both the Fletcher–Reeves version of the nonlinear conjugate gradient method and a polynomial preconditioner type based on block two-stage methods. Several strategies of parallelization and different storage formats for sparse matrices are discussed. The reported numerical experiments analyze the behavior of these algorithms working in a fine grain parallel environment compared with a thread-based environment.

1. Introduction

Graphics Processing Units (GPUs) have become a powerful many-core processor and useful tool within the computational science community. The massively parallel architecture offers high performance in many computing applications when the algorithms map well to the characteristics of the GPU.

The conjugate gradient (CG) algorithm [10] is an efficient method for solving linear systems with symmetric positive definite matrices. The performance of this algorithm can be improved by considering the preconditioned conjugate gradient (PCG) method. Its efficiency and robustness have been proved for a wide range of applications. There are several works about this kind of linear algorithms for GPUs. In [16] a sparse matrix solver for the conjugate gradient method on GPUs is described. On the other hand, in [5] an efficient implementation of a Jacobi preconditioned conjugate gradient algorithm on GPUs is developed. Recently, an SSOR preconditioned conjugate gradient algorithm on GPUs is analyzed in [9]. The authors focus on the numerical solution of the generalized Poisson equation and they notice that the proposed parallel implementation is significantly better than the Jacobi preconditioner implemented in [5]. Moreover, as compared to the CPU implementation of the CG algorithm, their GPU preconditioned conjugate gradient implementation is up to 10 times faster; other related works can be found e.g., in [3,8,18].

Focused on the problem of solving nonlinear systems, there are some works that deal with parallel solvers on GPU. At present, techniques for solving nonlinear systems on GPUs using wavelet analysis are developed. Moreover, parallel solvers for nonlinear systems in Bernstein form based on subdivision and the Newton–Raphson methods are proposed; see e.g., [6,27] and the references cited therein. In this work we develop parallel algorithms for solving nonlinear systems based on the CG method.

The CG method for linear systems can be extended for nonlinear systems following the Fletcher–Reeves version [7]. For this purpose, consider the problem of solving the mildly nonlinear system

$$Ax = \Phi(x), \tag{1}$$

where $A \in \Re^{n\times n}$ is a symmetric positive definite matrix and $\Phi : \Re^n \rightarrow \Re^n$ is a nonlinear diagonal mapping, i.e., the ith component ϕ_i of ϕ is a function only of the ith component x_i of x. Let $\Psi : \Re^n \rightarrow \Re$ be a nonlinear mapping and consider $\langle x, y\rangle = x^T y$ the inner product in $\Re^n$. The minimization problem of finding $x \in \Re^n$ such that $J(x) = \min_{y\in\Re^n} J(y)$, where $J(x) = \frac{1}{2}\langle Ax, x\rangle - \Psi(x)$, is equivalent to find $x \in \Re^n$ such that $F(x) = Ax - \Phi(x) = 0$, where $\Phi(x) = \Psi'(x)$. The Fletcher–Reeves version [7] of the nonlinear conjugate gradient method (NLCG) is an effective approach for

☆ This research has been supported by the Spanish Ministry of Science and Innovation under grant number TIN2011-26254.

* Corresponding author.
E-mail addresses: vgaliano@umh.es (V. Galiano), hmigallon@umh.es (H. Migallón), violeta@dccia.ua.es (V. Migallón), jpenades@dccia.ua.es (J. Penadés).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.10.016

solving the nonlinear system (1), by considering the connection with this minimization problem:

Algorithm 1 (*Fletcher–Reeves Nonlinear Conjugate Gradient*)**.**

```
Given an initial vector $x^{(0)}$
  $r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
  $p^{(0)} = r^{(0)}$
  For $i = 0, 1, \dots,$ until convergence
      $\alpha_i =\!\rightarrow$ see below
      $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
      $r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
      Convergence test
      $\beta_{i+1} = -\frac{\langle r^{(i+1)}, r^{(i+1)}\rangle}{\langle r^{(i)}, r^{(i)}\rangle}$
      $p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$
```

Note that, in Algorithm 1, α_i is chosen to minimize the associated functional J in the direction $p^{(i)}$. This is equivalent to solve the one dimensional zero-point problem $\frac{dJ(x^{(i)}+\alpha_i p^{(i)})}{d\alpha_i} = 0$. Using the Newton method for solving the zero-point problem for α_i, it obtains $\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$, where

$$\delta^{(k)} = \frac{\alpha_i^{(k)} \left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle r^{(i)}, p^{(i)}\right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i^{(k)} p^{(i)}), p^{(i)}\right\rangle}{\left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i^{(k)} p^{(i)}) p^{(i)}, p^{(i)}\right\rangle}.$$

Note that in order to obtain $\delta^{(k)}$, the inner products $\langle Ap^{(i)}, p^{(i)}\rangle$ and $\langle r^{(i)}, p^{(i)}\rangle$ can be computed once at the first Newton iteration. Moreover $Ap^{(i)}$ is available from the conjugate gradient iteration.

In order to generate efficient algorithms to solve the nonlinear system (1), in [17] we designed a parallel version of Algorithm 1 and a parallel nonlinear preconditioned conjugate gradient algorithm, based on both Algorithm 1 and a polynomial preconditioner type based on the block two-stage methods [4]. The proposed parallel algorithms were analyzed on multicore architectures using an OpenMP model [23]. In this paper we have developed the implementation of these nonlinear algorithms on GPUs. The Fletcher–Reeves version of the nonlinear conjugate gradient algorithm and its preconditioned method are based on a set of operations; the most important of which are the sparse matrix vector product (SpMV) and the inner (or dot) product, in addition to other vector computations. We discuss several strategies to compute these operations and we consider different storage formats for sparse matrices. Furthermore, in the preconditioned method, incomplete LU factorizations are used in order to obtain the inner splittings of the block two-stage method, therefore we analyze several strategies taking into account that there is no fine grain inherent parallelism in the LU solver.

The algorithms described here have been implemented on an Intel Core 2 Quad Q6600 and an NVIDIA GTX 280 GPU. In order to analyze the behavior of these algorithms we have considered a nonlinear elliptic partial differential equation known as the Bratu problem [2]. In Section 2 we describe the preconditioned conjugate gradient algorithm and the constructed preconditioners. Some concepts about GPU architecture and its parallel programming are given in Section 3. In Section 4 we review the sparse matrix storage formats used in this work and in Section 5 we explain how the basic operations have been implemented in order to perform the algorithms described in Sections 1 and 2. In Section 6, we display the numerical results obtained using CUDA on a GPU and we compare these results with those obtained on the shared memory platform using OpenMP. Furthermore, a mixed model (using OpenMP and CUDA) is considered in order to exploit the characteristics of both parallel systems. Finally, concluding remarks are presented in Section 7.

2. Nonlinear preconditioned conjugate gradient method

Preconditioning is a technique for improving the condition number (cond) of a matrix. Suppose that M is a symmetric positive definite matrix that approximates A, but is easier to invert. We can solve $Ax = \Phi(x)$ indirectly by solving $M^{-1}Ax = M^{-1}\Phi(x)$. If $\mathrm{cond}(M^{-1}A) \ll \mathrm{cond}(A)$ we can iteratively solve $M^{-1}Ax = M^{-1}\Phi(x)$ more quickly than the original problem. In this case we obtain the following nonlinear preconditioned conjugate gradient algorithm (NLPCG).

Algorithm 2 (*Nonlinear Preconditioned Conjugate Gradient*)**.**

```
Given an initial vector $x^{(0)}$
  $r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
  Solve $Ms^{(0)} = r^{(0)}$
  $p^{(0)} = s^{(0)}$
  For $i = 0, 1, \dots,$ until convergence
      $\alpha_i =\!\rightarrow$ see Algorithm 1
      $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
      $r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
      Solve $Ms^{(i+1)} = r^{(i+1)}$
      Convergence test
      $\beta_{i+1} = -\frac{\langle s^{(i+1)}, r^{(i+1)}\rangle}{\langle s^{(i)}, r^{(i)}\rangle}$
      $p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$
```

Since the auxiliary system $Ms = r$ must be solved at each conjugate gradient iteration, this system needs to be easily solved. Moreover, in order to obtain an effective preconditioner, it wants M to be a good approximation to A. One of the general preconditioning techniques for solving linear systems is the use of the truncated series preconditioning [1]. These preconditioners consist of considering a splitting of the matrix A as

$$A = P - Q \tag{2}$$

and performing m steps of the iterative procedure defined by this splitting toward the solution of $As = r$, choosing $s^{(0)} = 0$. It is well known that the solution of the auxiliary system $Ms = r$ is effected by $s = (I + R + R^2 + \cdots + R^{m-1})P^{-1}r$, where $R = P^{-1}Q$ and the preconditioning matrix is $M_m = P(I + R + R^2 + \cdots + R^{m-1})^{-1}$, cf. [1]. In order to obtain the preconditioners we use m steps of block two-stage methods toward the solution of $As = r$, choosing $s^{(0)} = 0$. More specifically, suppose that A is partitioned into $p \times p$ blocks, with square diagonal blocks of order n_j, $\sum_{j=1}^{p} n_j = n$, such that system (1) can be written as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \Phi_1(x) \\ \Phi_2(x) \\ \vdots \\ \Phi_p(x) \end{bmatrix}, \tag{3}$$

where x and $\Phi(x)$ are partitioned according to the size of the blocks of A. Let us consider the splitting (2) such that P consists of the diagonal blocks of A in (3), that is $P = \mathrm{diag}(A_{11}, \dots, A_{pp})$. Let us consider the splittings $A_{jj} = B_j - C_j$, $1 \le j \le p$, then, to solve the auxiliary system $Ms = r$ of Algorithm 2, the following algorithm is established.

Algorithm 3 (*Block Two-Stage*)**.** Given an initial vector $s^{(0)} = ((s_1^{(0)})^T, (s_2^{(0)})^T, \dots, (s_p^{(0)})^T)^T$ and a sequence of numbers of inner iterations $q(j)$, $1 \le j \le p$

```
For $l = 1, 2, \ldots,$ until convergence
    For $j = 1, 2, \ldots, p$
        $y_j^{(0)} = s_j^{(l)}$
        For $k = 1$ to $q(j)$
            $B_j y_j^{(k)} = C_j y_j^{(k-1)} + (Qs^{(l-1)} + r)_j$
        $s^{(l)} = \left((y_1^{(q(1))})^T, (y_2^{(q(2))})^T, \ldots, (y_p^{(q(p))})^T\right)^T$
```

In order to analyze the NLPCG method, we have considered in Algorithm 3 an incomplete LU factorization of each matrix A_{jj}, $j = 1, 2, \ldots, p$, that is, the splittings $A_{jj} = B_j - C_j$ are constructed by setting $B_j = L_j U_j$ and $C_j = R_j$, and at each lth step perform, for each j, $q(j)$ inner iterations of the iterative procedure defined by this splitting. Let us denote by ILU(S) the incomplete LU factorization associated with the zero pattern subset S of $S_n = \{(i,j) : i \neq j, 1 \leq i,j \leq n\}$. In particular, when $S = \{(i,j) : a_{ij} = 0\}$, the incomplete factorization with zero fill-in, known as ILU(0), is obtained. To improve the quality of the factorization, many strategies for altering the pattern have been proposed. In the "level of fill-in" factorizations [12], ILU(κ), $\kappa \geq 0$, a level of fill-in is recursively attributed to each fill-in position from the levels of its parents. Then, the positions of level lower than κ are removed from S. In the experiments reported here we have used these ILU(κ) factorizations for the matrices A_{jj}, $j = 1, 2, \ldots, p$, previously defined; see [17] for a more detailed explanation of the algorithms treated here and their OpenMP implementations.

3. GPU architecture — CUDA parallel programming

The GPU architecture is based on a set of multiprocessor units called streaming multiprocessors (SM), each one containing a set of processor cores called streaming processors (SP). CUDA is a heterogeneous computing model that involves both the CPU and the GPU. In the CUDA parallel programming model [19,22], an application consists of a sequential host program, that may execute parallel programs known as kernels on a parallel device, i.e., on a GPU. Note that the CPU could be a multicore processor running an OpenMP model program; in this case only one core can call a kernel, i.e., kernel calls must be serialized. A kernel is an SPMD (Single Program Multiple Data) computation that is executed using a potentially large number of parallel threads. Each thread runs the same scalar sequential program. The programmer organizes the threads of a kernel into a grid of thread blocks. The threads of a given block can cooperate among themselves using barrier synchronization. The main memories available in GPUs are: the global memory, which has the highest latency; the constant and the texture memories, which are read only memories; shared memory and registers, both are on-chip memories, shared memory is owned by a block and registers are owned by a thread. The constant and the texture memories have on-chip cache. In this work the global and shared memories have been used.

Thread creation, scheduling, and management are performed entirely in hardware. For example, the GTX 280 GPU contains 30 multiprocessors, and it can work with a maximum of 30 K concurrent threads. In order to manage efficiently this large number of threads, the GPU employs an SIMT (Single Instruction Multiple Thread) architecture [14,19] in which the threads of a block are executed in groups of 32 called warps. A warp executes a single instruction at a time across all its threads. The threads of a warp are free to follow their own execution. The thread execution may diverge, however, it is substantially more efficient for threads to follow the same execution path for the bulk of the computation.

4. Sparse matrix storage formats

There are a multitude of sparse matrix representations, each one with different storage requirements, computational characteristics, and methods of accessing and manipulating matrix entries. In the context of sparse matrix vector product on GPU, we have only considered common sparse matrix storage formats suitable to obtain a good behavior for computing on GPU. Concretely, we have used the Compressed Row Storage (CRS) format, the ELLPACK (or ITPACK) format [11], and the ELLPACK-R format proposed in [25]. A discussion of these storage formats for the sparse matrix vector product (SpMV) on GPUs can be found in [26,25].

The Compressed Row Storage (CRS) format is a popular, general-purpose sparse matrix representation. Assuming we have a nonsymmetric sparse matrix A, the CRS format stores the matrix on three vectors: one for floating point numbers and other two for integers. The floating point vector stores the values of the nonzero elements of the matrix A, following a row-wise method. One of the integer vectors stores the column indexes of the elements in the values vector. The other integer vector stores the locations in the values vector that start a row.

ELLPACK [11] was introduced as a format to compress a sparse matrix with the purpose of solving large sparse linear systems on vector computers. Note that there are some similarities between a vector architecture and the GPU architecture. This format stores the sparse matrix on two arrays, one for floating point numbers, to store the nonzero elements, and one for integers, to store the columns of every nonzero element. This format, as we have previously mentioned, is appropriate to compute operations with sparse matrices on vector architectures. However, as ELLPACK uses the same number of elements per row padding with zero elements, if the percentage of zeros is high and there is a very irregular location of entries in different rows, then the performance of the ELLPACK data structure decreases and storage requirements increase, see [26,25].

ELLPACK-R is a variant of the ELLPACK format introduced in [25] with the purpose of optimizing the sparse matrix vector product on GPUs. ELLPACK-R consists of two arrays following original ELLPACK and moreover, an additional integer array is included with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded.

5. GPU kernels

In this section, the primary kernels used in the implementation of the NLCG and NLPCG algorithms are explained. The kernels represented here perform mathematical operations used in the algorithms. Following Algorithms 1 and 2, the basic operations to implement the proposed methods are sparse matrix vector product, vector operations (included in level 1 BLAS [13]), inner product and LU solver (included in SPARSKIT [24] only for NLPCG method; see Section 6). In the rest of this section, we describe different ways of computing these basic operations and we analyze them in the context of our work, that is, in the context of solving nonlinear systems using the NLCG and NLPCG methods. Some optimizations will be considered in Section 6 from a global perspective with respect to the algorithms. Some of these operations are also available as kernels in several libraries as CUBLAS [20] or CUSPARSE [21]. In Section 6 our kernels are compared to those of these libraries.

5.1. Sparse matrix vector product

In order to compute the sparse matrix vector product (SpMV), we consider the different sparse matrix storage formats described in Section 4. Particularly, the kernel code to compute the SpMV, using CRS format, is not optimized. In order to optimize the code there are two ways, the first one is to use a storage format to optimize subsequent computation and the second one is to

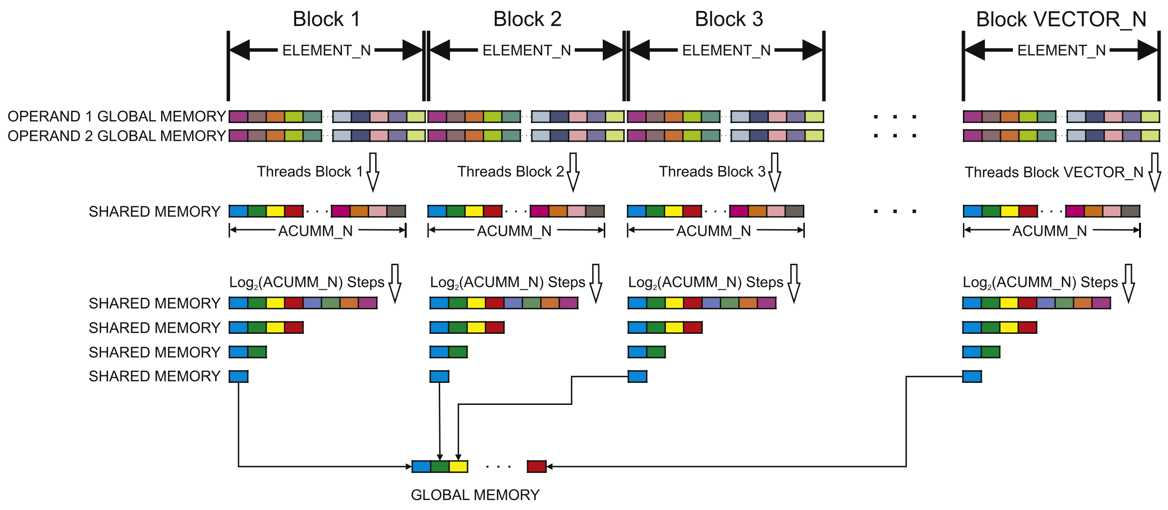


Fig. 1. Inner product according to the *NVIDIA CUDA C SDK*.

optimize the access to the GPU global memory modifying the thread mapping. However, in the second case the performed optimizations based on the CRS format are focused on matrices with higher nonzero elements per row than the matrices used in our work. Note that, in our test example the typical number of nonzero elements per row is 7 (see Section 6), and, for example, the optimizations presented in [15] need more than 32 nonzero elements per row. Consequently, in order to optimize the SpMV operation, we consider the use of ELLPACK and ELLPACK-R storage formats following [25]. Note that in both formats the memory access pattern is improved. Moreover, according to the mapping of threads in the computation of every row, we also consider the SpMV implementations proposed in [26] and based on ELLPACK-R format. In this proposal, T threads compute the ith element of the matrix-vector product accessing to the ith row of the matrix. The implementation of the SpMV following [25] is referred in the numerical experiments as ELLR, while the implementation of [26] is referred to as ELLR-T.

On the other hand, in order to compute the SpMV we also test the use of CUSPARSE [21]. CUSPARSE is a new library of GPU-accelerated sparse matrix routines for sparse/sparse and dense/sparse operations. Currently, the sparse matrix vector product is only supported in CRS format.

5.2. *Vector operations*

As we can see in Algorithms 1 and 2, the common vector operations, regardless of the reduction process, are the vector copy, the scalar vector product, the *axpy* operation and the nonlinear function computation of the nonlinear system. In order to optimize these operations, we try to group several operations into a single kernel. For example, the inner products needed for the computation of α as well as the inner products involved in the β computation are grouped in single kernels. On the other hand, CUBLAS [20] has been used to compute the basic operations with vectors. CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) using CUDA. Note that the use of CUBLAS does not allow to group several operations into a single kernel, however in our code we perform the *axpy* function and the vector copy function in the same kernel.

5.3. *Inner product*

The inner (or dot) product of a vector is a special operation in CUDA because it implies a reduction process. It is necessary to be able to use multiple thread blocks in order to process large vectors and keep busy all streaming multiprocessors (SM) of the GPU. For this purpose each thread block makes a reduction of a portion of the vector, but CUDA has no global synchronization to communicate partial results between thread blocks. To perform inner products we follow the proposal of *NVIDIA CUDA C SDK*, i.e., we compute *VECTOR_N* vectors of *ELEMENT_N* elements. *ELEMENT_N* must be a multiple of the warp size to meet alignment constraints of memory coalescing. One block computes the reduction of one or more portions of vector. In order to avoid synchronization processes, we work with a shared memory array of size *ACCUM_N* which acts as an accumulator. *ACCUM_N* must be a power of two and preferably a multiple of the warp size. Each thread computes an accumulator element through vectors with stride equal to *ACCUM_N*. To finish the kernel we perform a tree-like reduction of the results stored in the accumulator array, where synchronization processes between threads are necessary (see Fig. 1). Note that, in our implementation, CPU must complete the operation by computing the *VECTOR_N* partial results. The reduction of the *VECTOR_N* partial results could be performed in the CPU or in the global memory of the GPU. However, we have tested both options and we have observed that the use of the global memory of the GPU for this reduction does not increase the performance. This is due to the fact that this reduction does not need a lot of communications between CPU and GPU, and using the GPU we need to execute a kernel performing a reduction process of $\log_2$(*VECTOR_N*) steps. Note that the optimal value of *VECTOR_N* parameter is 128, therefore we must use a kernel with a single block with 64 threads. We have tested this process working on both global memory and shared memory, in both cases we obtain a slight decrease of the performance than when using the CPU.

On the other hand, CUBLAS also provides inner product, and taking into account that we work with the optimized version included in CUDA Toolkit 3.2 RC, we will not consider other optimizations.

5.4. *LU solver*

In the LU solver each computed element of the solution vector is needed to compute the next element. There is no inherent fine grain parallelism. We have tried various strategies to compute the LU solver on the GPU but efficiency has not been achieved. Remark that this procedure performs two loops without dependencies to compute each element of the solution vector. In the first strategy, we launch N blocks in one or more kernels, where N is the size

Kernels for NLCG	Kernels for NLPCG
Compute $Ax^{(0)}$	Compute $Ax^{(0)}$
Compute $r^{(0)}$ and $p^{(0)}$	Compute $r^{(0)}$
	Compute $s^{(0)}$ on the CPU
	Copy $s^{(0)}$ into $p^{(0)}$
Repeat until convergence	Repeat until convergence
Inner prod. needed for α (1st iter.)	Inner prod. needed for α (1st iter.)
Inner prod. needed for α (subseq. iter.)	Inner prod. needed for α (subseq. iter.)
Compute $x^{(i+1)}$ and copy $x^{(i)}$	Compute $x^{(i+1)}$ and copy $x^{(i)}$
Compute $Ap^{(i)}$	Compute $Ap^{(i)}$
Compute $r^{(i+1)}$ and copy $r^{(i)}$	Compute $r^{(i+1)}$ and copy $r^{(i)}$
	Compute $s^{(i+1)}$ on the CPU
Inner prod. needed for β	Inner prod. needed for β
Compute $p^{(i+1)}$	Compute $p^{(i+1)}$

Fig. 2. Kernels sequence for the NLCG and NLPCG algorithms.

of the system, each thread computes its partial computation of both loops, performing a reduction process to complete the loops. In the second one, following the partitioning showed in (3), the computation of each block solution is assigned to each SM of the GPU. Therefore we perform a partitioning based on the number of SMs of the GPU, 30 on the GTX 280. On the other hand, strategies that group independent rows into levels in order to sequentially iterate across the constructed levels, as the one presented in [18], entail a sequential computation due to the sparsity pattern of the LU in our matrix problem. Hence, the parallelism involved by the preconditioner algorithm (Algorithm 3) has been exploited with the multicore CPU architecture. However, this parallelism becomes somewhat ineffective due to CPU–GPU communications. Due to the structure of (3) and assuming that p is the number of cores, the number of iterations of the NLPCG method, in a multicore, increases with the number of cores; therefore the amount of communications between CPU and GPU increases. In Section 6 we will show results using CUDA and OpenMP in multicore CPU. Note that in the NLCG method the communications between CPU and GPU are reduced to some scalars and the arrays to complete reduction operations.

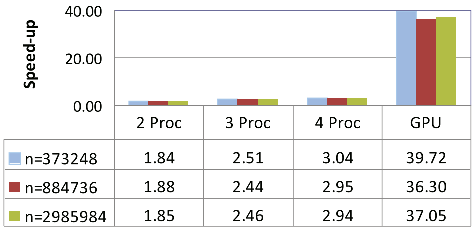
Fig. 2 summarizes the kernels used in each algorithm showing the involved grouped operations according to the structure of Algorithms 1 and 2. We note that the same kernel is used in both algorithms for the same operation, except for the computation of β and $p^{(i+1)}$.

6. Numerical experiments

In order to illustrate the behavior of the NLCG and NLPCG methods, we have run both algorithms on a multicore computer Intel Core 2 Quad Q6600, 2.4 GHz, with 4 GB of RAM and 8 MB of L2 Cache Memory, called SULLI. The operating system in SULLI is Ubuntu 9.04 (Jaunty Jackalope) for 64 bit systems. The GPU is an NVIDIA GeForce GTX 280 connected to SULLI. The CUDA kernels were compiled using the NVIDIA CUDA compiler (nvcc) from CUDA Toolkit 3.2 RC. As our illustrative example we have considered a nonlinear elliptic partial differential equation, known as the Bratu problem. In this problem, heat generation from a combustion process is balanced by heat transfer due to conduction. The three-dimensional model problem is given as $\nabla^2 u - \lambda e^u = 0$, where u is the temperature and λ is a constant known as the Frank-Kamenetskii parameter; see e.g., [2]. There are two possible steady-state solutions to this problem for a given value of λ. One solution is close to $u = 0$ and it is easy to obtain. A starting point near to the other solution is needed to converge to it. For our model case, we consider a 3D cube domain Ω of unit length and $\lambda = 6$. To solve this problem using the finite difference method, we consider a grid in Ω of d^3 nodes. This discretization yields a nonlinear system of the form $Ax = \Phi(x)$, where $\Phi : \Re^n \rightarrow \Re^n$ is a nonlinear diagonal mapping, i.e., the ith component Φ_i of Φ is a function only of the ith component of x. The matrix A is a sparse matrix of order $n = d^3$ and the typical number of nonzero elements per row of this matrix is seven, with fewer in rows corresponding to boundary points of the physical domain.

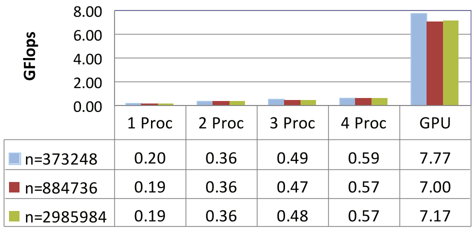
The performed analyses are based on the run-times and the number of floating point operations per second (Flops) measured on a GeForce GTX 280 compared with the parallel run-times and Flops measured on SULLI using OpenMP. First we present results for the NLCG method with problems of various sizes. In Fig. 3(a) we show the speed-up using OpenMP and different number of cores in SULLI, and the speed-up when the NLCG method is computed on the NVIDIA GeForce GTX 280 GPU, managed from one core of SULLI. In addition, Fig. 3(b) compares this method in terms of GFLOPS on both architectures. We obtain a good speed-up with OpenMP using the available cores, but not comparable with the speed-up obtained with GPU, greater than 36. These results confirm a good interaction between the NLCG algorithm and a GPU computing platform. Note that, on the GPU, the NLCG method is about 12–13 times faster than using four cores of the CPU, achieving about 7 GFlops.

In order to call a CUDA kernel, the number of threads of each block must be established taking into account that the maximum number of threads is 512. Therefore, the number of blocks in each grid depends on the number of required threads. For example, if each block has 512 threads, for calling the *axpy* kernel working with a system of size $n = 373{,}248$, a grid with 729 blocks is required. On the other hand, for calling the inner product kernel, the number of blocks and the number of threads in each block should be selected in order to obtain the best performance. We have analyzed the behavior of the NLCG algorithm with respect to the number of threads in each block and the influence of *ACCUM_N*, that is, the size of the shared memory arrays that



	2 Proc	3 Proc	4 Proc	GPU
🟦 n=373248	1.84	2.51	3.04	39.72
🟥 n=884736	1.88	2.44	2.95	36.30
🟩 n=2985984	1.85	2.46	2.94	37.05

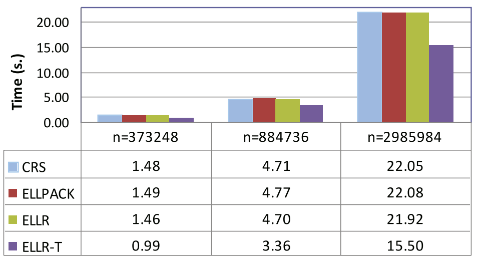
(a) Speed-up.



	1 Proc	2 Proc	3 Proc	4 Proc	GPU
🟦 n=373248	0.20	0.36	0.49	0.59	7.77
🟥 n=884736	0.19	0.36	0.47	0.57	7.00
🟩 n=2985984	0.19	0.36	0.48	0.57	7.17

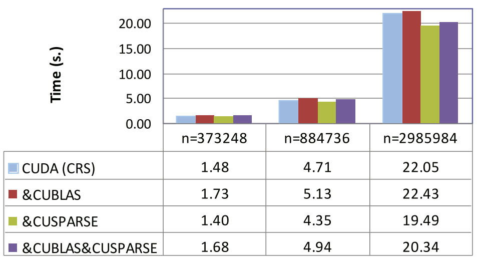
(b) GFlops.

Fig. 3. NLCG method.



	n=373248	n=884736	n=2985984
🟦 CRS	1.48	4.71	22.05
🟥 ELLPACK	1.49	4.77	22.08
🟩 ELLR	1.46	4.70	21.92
🟪 ELLR-T	0.99	3.36	15.50

(a) Comparison sparse matrix storage formats.



	n=373248	n=884736	n=2985984
🟦 CUDA (CRS)	1.48	4.71	22.05
🟥 &CUBLAS	1.73	5.13	22.43
🟩 &CUSPARSE	1.40	4.35	19.49
🟪 &CUBLAS&CUSPARSE	1.68	4.94	20.34

(b) Comparison CUBLAS and/or CUSPARSE.

Fig. 4. NLCG method.

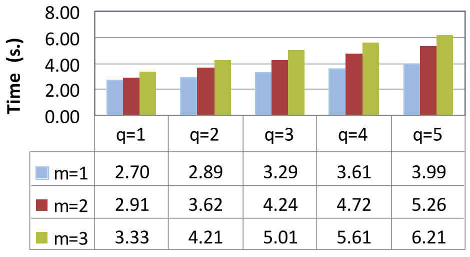
act as accumulators. Although the influence of the number of threads in each block and *ACCUM_N* is not critical, the best performance is obtained using 128 or 256 threads in each block and *ACCUM_N* $= 128$.

In Fig. 4(a) we analyze the performance of the different storage formats explained in Section 4. The sparse matrix storage format used changes the code to compute the SpMV operation (see Section 5.1). The best results are obtained using ELLPACK-R format (ELLR and ELLR-T). The algorithms for computing SpMV using ELLPACK-R do not include flow control instructions that serialize the execution of a warp of 32 threads and they allow coalesced matrix data access. Note that ELLPACK-R format is the sparse matrix storage format with highest memory requirement when compared with the other discussed formats. This is due to the use of a vector of integers in order to store the number of nonzero elements of each row and to avoid the flow control inside the iterative loop for computing each element. The results shown in Fig. 4(a) have been obtained using optimal values for all the parameters. In the case of the ELLR-T implementation, for our problem, the best performance has been achieved with $T = 8$ and using a thread block size of 256 (see [26] for a detailed description of these parameters). In [25] it is confirmed that, for matrices with high sparsity pattern having almost the same number of nonzero elements per row, the performance improvement obtained with ELLR for the SpMV, is not significant compared with the use of CRS and ELLPACK formats. In fact, for these three cases, the SpMV represents, in our test, problems of about the 50% of the total solving time, and the analysis of the performance (GFlops) of the SpMV kernels of the algorithm shows similar results for these storage formats (between 2.3 and 2.6 GFlops). However, the ELLR-T implementation outperforms the other approaches. In this case the SpMV represents about the 30% of the total time, achieving between 5.6 and 5.8 GFlops.

To conclude the analysis of the NLCG method, we present in Fig. 4(b) results using the CUBLAS and CUSPARSE libraries, included in the CUDA Toolkit 3.2 RC. We analyze the use of CUBLAS and the use of CUSPARSE separately, and the use of both together. First, when we only use CUBLAS library the results are worse than the results using only the CUDA API (with CRS format). This is due to the optimizations performed by grouping operations, which can not be performed using CUBLAS. In addition, reduction operations may be affected by the parameters. In contrast, the use of CUSPARSE leads to a slight improvement. In this case the SpMV kernels of the algorithm achieve between 2.5 and 3.1 GFlops. On the other hand the *axpy* operations achieve more than 4 GFlops and the inner product operations between 8.5 and 11.5 GFlops. However, the best performance is obtained in the iterative procedure to compute α achieving between 15.2 and 19.9 GFlops. Remark that CUBLAS and CUSPARSE libraries hide to the user the setting of parameters, both for calling kernels and in the reduction operations.

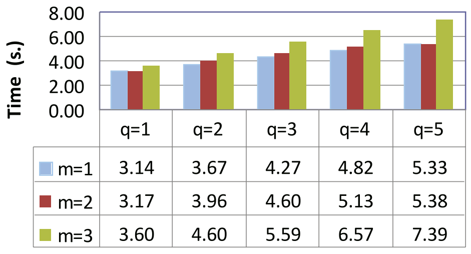
In order to analyze the NLPCG method, first we present results showing the behavior of this method respect to its characteristic parameters, which are the number of outer iterations, the number of inner iterations of the block two-stage method and the level of fill-in of the incomplete LU factorizations. Note that, as we have mentioned in Section 5.4, the LU solver is computed in the multicore, where p is the number of cores. Initially, we consider only one block, thus the method uses one core and the GPU. In Fig. 5 we present results varying the number of inner iterations q and the number of outer iterations m, for a system of size equal to 373,248 and level of fill-in equal to 0 and equal to 1. Fig. 5 shows that, for the matrix of size 373,248, the best results are obtained using $q = 1$ and $m = 1$. More specifically, at each nonlinear iteration of the NLPCG algorithm, only one step ($m = 1$) of the truncated series preconditioner defined by Algorithm 3 is used, performing $q = 1$ inner iterations of the iterative procedure defined by the incomplete LU factorization (see Section 2). All performed experiments have shown analogous results to those obtained in a multicore architecture, where the best results were obtained with small values of q and m (see [17]).

Fig. 6 presents results using 2 cores and the GPU to solve a system of size 884,736. The results correspond to a level of fill-in equal to 0. Since the LU solver is performed at the CPU, using more than one core can benefit the algorithm as it is shown in Fig. 7. However, it is important to remark that when optimal values are used, it should not be used more than one core, because the increase in the number of global iterations required does not allow



	q=1	q=2	q=3	q=4	q=5
m=1	2.70	2.89	3.29	3.61	3.99
m=2	2.91	3.62	4.24	4.72	5.26
m=3	3.33	4.21	5.01	5.61	6.21

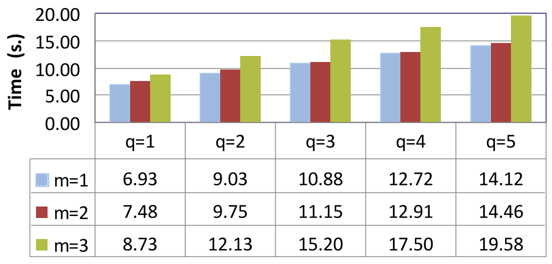
(a) ILU(0).



	q=1	q=2	q=3	q=4	q=5
m=1	3.14	3.67	4.27	4.82	5.33
m=2	3.17	3.96	4.60	5.13	5.38
m=3	3.60	4.60	5.59	6.57	7.39

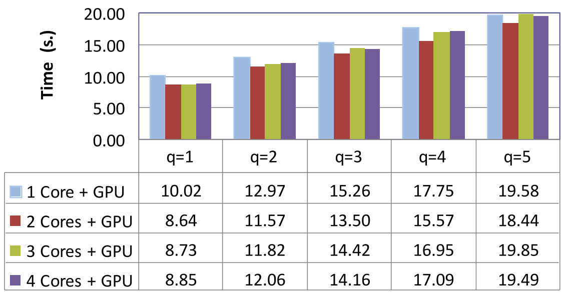
(b) ILU(1).

Fig. 5. NLPCG, 1 core + GPU, $n = 373{,}248$.



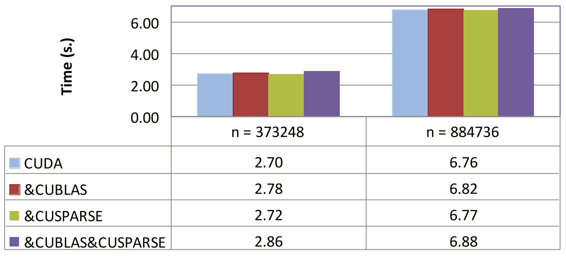
	q=1	q=2	q=3	q=4	q=5
m=1	6.93	9.03	10.88	12.72	14.12
m=2	7.48	9.75	11.15	12.91	14.46
m=3	8.73	12.13	15.20	17.50	19.58

Fig. 6. NLPCG, 2 cores + GPU, $n = 884{,}736$, ILU(0).



	q=1	q=2	q=3	q=4	q=5
1 Core + GPU	10.02	12.97	15.26	17.75	19.58
2 Cores + GPU	8.64	11.57	13.50	15.57	18.44
3 Cores + GPU	8.73	11.82	14.42	16.95	19.85
4 Cores + GPU	8.85	12.06	14.16	17.09	19.49

Fig. 7. NLPCG, $n = 884{,}736$, $m = 2$, ILU(1).



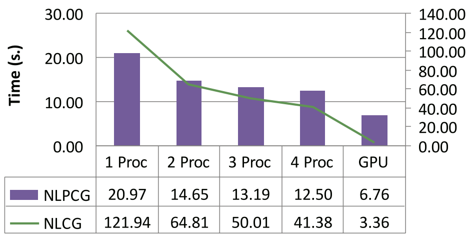
	n = 373248	n = 884736
CUDA	2.70	6.76
&CUBLAS	2.78	6.82
&CUSPARSE	2.72	6.77
&CUBLAS&CUSPARSE	2.86	6.88

Fig. 8. NLPCG, 1 core + GPU, $m = 1$, $q = 1$, ILU(0).

an improvement. The conclusions about the optimal value of the number of threads in each block have been analogous to those obtained for the NLCG method. On the other hand, the behavior of the NLPCG algorithm is not affected by the value of *ACUMM_N*. Fig. 8 shows that the use of CUBLAS in the NLPCG method slightly increases the computational times, as we have also seen in the NLCG method. Moreover, the NLPCG method is not affected by using CUSPARSE. It is due to the small number of iterations of the NLPCG method, which is nearly 10 times lower than the iterations needed by the NLCG method. Finally, from the analysis of Fig. 9 it is deduced that the speed-up using GPU is greater than the speed-up using four cores of the CPU. Moreover, as compared to the CPU implementation of the NLCG algorithm, our GPU implementation of the NLPCG is up to 18–20 times faster. Nevertheless the speed-up of the NLPCG method is not comparable with the one obtained by the NLCG method, which is above 36. This is due to the need to run the LU solver on the CPU, and the decrease of the work performed on the GPU. In addition, the communications between CPU and GPU are increased. Note that the percentage of the execution time required by the LU solver in the NLPCG algorithm (using one core of the CPU) is about 40%–50%. This percentage increases up to 60%–70% when CPU–GPU communications are also considered. Clearly, both methods present a very different inherent parallelism, therefore their performance on two different parallel architectures is also different. However, as can be expected, the best execution times of both methods are always obtained when the algorithms make use of the GPU.

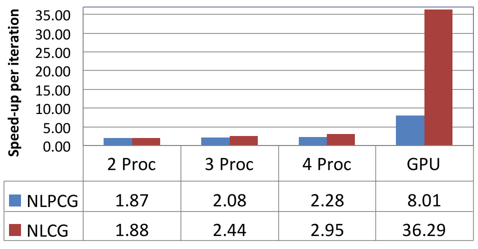
7. Conclusions

We have developed the Fletcher–Reeves version of the nonlinear conjugate gradient method and we have applied a polynomial preconditioner based on two-stage methods. We have analyzed both methods using a multicore OpenMP model, a GPGPU model and a mixed model in order to exploit both parallel systems. We have analyzed the proposed algorithms in order to identify the main operations, and we have implemented some optimizations and tested some libraries in order to perform these operations optimally. CUBLAS and CUSPARSE libraries offer a good performance, and the sparse matrix format should be chosen according to the parallel architecture, being ELLPACK-R and specifically its ELLR-T implementation, the most efficient format.



	1 Proc	2 Proc	3 Proc	4 Proc	GPU
NLPCG	20.97	14.65	13.19	12.50	6.76
NLCG	121.94	64.81	50.01	41.38	3.36

(a) Total execution time.



	2 Proc	3 Proc	4 Proc	GPU
NLPCG	1.87	2.08	2.28	8.01
NLCG	1.88	2.44	2.95	36.29

(b) Speed-up per iteration.

Fig. 9. Comparison NLCG and NLPCG on CPU and GPU, $n = 884{,}736$.

We would like to point out that the use of the GPU improves the results obtained using any of the proposed methods. On the other hand, the NLCG method exploits better the parallelism offered by the GPU than the NLPCG method.

References

[1] L. Adams, *M*-step preconditioned conjugate gradient methods, SIAM J. Sci. Stat. Comput. 6 (1985) 452–462.
[2] B.M. Averick, R.G. Carter, J.J. More, G. Xue, The MINPACK-2 test problem collection, Technical Report MCS-P153-0692, Mathematics and Computer Science Division, Argonne, 1992.
[3] J. Bolz, I. Farmer, E. Grinspun, P. Schröder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, ACM Trans. Graph. 22 (3) (2003) 917–924.
[4] R. Bru, V. Migallón, J. Penadés, D.B. Szyld, Parallel, synchronous and asynchronous two-stage multisplitting methods, Electron. Trans. Numer. Anal. 3 (1995) 24–38.
[5] L. Buatois, G. Caumon, B. Lévy, Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emergent Distrib. Syst. 24 (3) (2009) 205–223.
[6] L. Dechevsky, B. Bang, J. Gundersen, A. Lakså, A.R. Kristoffersen, Solving non-linear systems of equations on graphic processing units, Lect. Notes Comput. Sci. 5910 (2010) 719–729.
[7] R. Fletcher, C. Reeves, Function minimization by conjugate gradients, The Comput. J. 7 (1964) 149–154.
[8] G.A. Gravvanis, C.K. Filelis-Papadopoulos, K.M. Giannoutakis, Solving finite difference linear systems on GPUs: CUDA parallel explicit preconditioned biconjugate conjugate gradient type methods, J. Supercomput. (2011) doi:10.1007/s11227-011-0619-z.
[9] R. Helfenstein, J. Koko, Parallel preconditioned conjugate gradient algorithm on GPU, J. Comput. Appl. Math. (2011) doi:10.1016/j.cam.2011.04.025.
[10] M.R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, J. Res. Nat. Bur. Stand. 49 (1952) 409–436.
[11] D.R. Kincaid, D.M. Young, A brief review of the ITPACK project, J. Comput. Appl. Math. 24 (1–2) (1998) 121–127.
[12] H.P. Langtangen, Conjugate gradient methods and ILU preconditioning of nonsymmetric matrix systems with arbitrary sparsity patterns, Internat. J. Numer. Methods Fluids 9 (1989) 213–233.
[13] C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, Basic linear algebra subprograms for FORTRAN usage, ACM Trans. Math. Software 5 (1979) 308–323.
[14] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro 28 (2) (2008) 39–55.
[15] M. Manikandan, R. Bordawekar, Optimizing sparse matrix-vector multiplication on GPUs, IBM Research Report RC24704, 2008.
[16] D. Michels, Sparse-matrix-CG-solver in CUDA, in: Proceedings of the 15th Central European Seminar on Computer Graphics, 2011.
[17] H. Migallón, V. Migallón, J. Penadés, Parallel nonlinear conjugate gradient algorithms on multicore architectures, in: J. Vigo (Ed.), Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering, 2009, pp. 689–700.
[18] M. Naumov, Parallel solution of sparse triangular linear in the preconditioned iterative methods on the GPU, NVIDIA Technical Report NVR-2011-001, 2011, http://research.nvidia.com/sites/default/files/publications/nvr-2011-001.pdf.
[19] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, ACM Queue 6 (2) (2008) 40–53.
[20] NVIDIA Corporation, CUDA CUBLAS Library, Document PG-05326-032_V01, 2010, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUBLAS_Library.pdf.
[21] NVIDIA Corporation, CUDA CUSPARSE Library, Document PG-05329-032_V01, 2010, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUSPARSE_Library.pdf.
[22] NVIDIA Corporation, NVIDIA CUDA C Programming Guide, Version 3.2, 2010, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf.
[23] OpenMP official site, 2008, http://openmp.org.
[24] Y. Saad, SPARSKIT: a basic tool kit for sparse matrix computation, http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html.
[25] F. Vázquez, J.J. Fernández, E.M. Garzón, A new approach for sparse matrix vector product on NVIDIA GPUs, Concurrency Computat.: Pract. Exp. 23 (2011) 815–826.
[26] F. Vázquez, G. Ortega, J.J. Fernández, E.M. Garzón, Improving the performance of the sparse matrix vector product with GPUs, in: 10th IEEE International Conference on Computer and Information Technology, CIT2010, 2010, pp. 1146–1151.
[27] F. Wei, J. Feng, H. Lin, GPU-based parallel solver via Kantorovich theorem for the nonlinear Bernstein polynomial system, Comput. Math. Appl. 62 (6) (2011) 2506–2517.

V. Galiano is a Professor of the Physics and Computer Architecture Department at the Miguel Hernandez University in Elche (Spain). He received the Ing. grad. from the Polytechnic University of Valencia and the Ph.D. degree in Computer Science in 2007. His main research interests include parallel algorithms for heterogeneous platforms for solving linear and nonlinear systems, high level interface design in parallel libraries for using them in meteorological and biological applications and parallel simulations for modeling electronic systems.

H. Migallón is a Professor of the Physics and Computer Architecture Department at the Miguel Hernandez University in Elche (Spain). He received the Degree in Physics and Electronic Eng. from the University of Valencia. He is member of the "Architecture and Computer Technology" at Miguel Hernández University and the "High Performance Computing and Parallelism" at the University of Alicante research groups. His main research interests include parallel high level interfaces for heterogeneous platforms, parallel algorithms for solving linear and nonlinear systems, parallel algorithms for image processing and parallel simulations for modeling electronic systems.

V. Migallón is Full Professor of the Computing Science and Artificial Intelligence Department and a research member of the University Institute for Computing Research, both at the University of Alicante (Spain). She received the Ph.D. degree in Computer Science in 1993. She is a member of the "High Performance Computing and Parallelism" research group at the University of Alicante which led from its foundation in 1991 until 2002. Her main research interests are scheduling techniques and parallel algorithms for heterogeneous platforms for solving linear and nonlinear systems and high level interface design, which simplifies programming in parallel environments.

J. Penadés is Full Professor of the Computing Science and Artificial Intelligence Department and a research member of the University Institute for Computing Research, both at the University of Alicante (Spain). He received the Ph.D. degree in Computer Science in 1993. His main research interests are scheduling techniques and parallel algorithms for heterogeneous platforms for solving linear and nonlinear systems and high level interface design, which simplifies programming in parallel environments. He currently leads the "High Performance Computing and Parallelism" research group in Alicante.