Paper 24

©Civil-Comp Press, 2011
Proceedings of the Second International Conference on
Parallel, Distributed, Grid and Cloud Computing for Engineering,
P. Iványi and B.H.V. Topping, (Editors),
Civil-Comp Press, Stirlingshire, Scotland

# GPU-Based Parallel Nonlinear Conjugate Gradient Algorithms

**V. Galiano[1], H. Migallón[1], V. Migallón[2] and J. Penadés[2]**
**[1] Department of Physics and Computer Architectures**
**University Miguel Hernández, Elche, Alicante, Spain**
**[2] Department of Computer Science and Artificial Intelligence**
**University of Alicante, Spain**

## Abstract

In this paper 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.

**Keywords:** GPGPU, GPU libraries, multicore, nonlinear conjugate gradient algorithms, parallel preconditioners, ILU factorizations, two-stage methods, Bratu problem.

## 1 Introduction

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 function with certain local smoothness properties. 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

$$J(x) = \min_{y\in\Re^n} J(y), \tag{2}$$

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 $\Phi(x) = \Psi'(x)$.

An effective approach for solving nonlinear system (1), by considering the connection with the minimization problem (2), is the Fletcher-Reeves version [1] of the nonlinear conjugate gradient method (NLCG), which takes the following form:

**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 $i = 0, 1, \ldots,$ 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, $\alpha_i$ is chosen to minimize the associated functional $J$ in the direction $p^{(i)}$. This is equivalent to solve the one dimensional zero-point problem $\frac{dJ(x^{(i)}+\alpha_i p^{(i)})}{d\alpha_i} = 0$. From the definition of $J$ it follows that

$$J(x^{(i)} + \alpha p^{(i)}) = \frac{1}{2} \left\langle A(x^{(i)} + \alpha_i p^{(i)}), x^{(i)} + \alpha_i p^{(i)} \right\rangle - \Psi(x^{(i)} + \alpha_i p^{(i)}).$$

Then a simple differentiation with respect to $\alpha_i$ yields

$$\frac{dJ(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i} = \alpha_i \left\langle Ap^{(i)}, p^{(i)} \right\rangle - \left\langle r^{(i)}, p^{(i)} \right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i p^{(i)}), p^{(i)} \right\rangle,$$

where $r^{(i)} = \Phi(x^{(i)}) - Ax^{(i)}$ is the nonlinear residual.

On the other hand, it is easy to see that the second derivative with respect to $\alpha_i$ takes the form

$$\frac{d^2 J(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i^2} = \left\langle Ap^{(i)}, p^{(i)} \right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i p^{(i)}) p^{(i)}, p^{(i)} \right\rangle.$$

Then, using the Newton method for solving the zero-point problem for $\alpha_i$, we obtain $\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$, where

$$\delta^{(k)} = \frac{dJ(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i}{d^2 J(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i^2} =$$

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

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), we have 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 [2]. The Fletcher-Reeves version of the developed nonlinear conjugate gradient algorithm is based on a set of routines; the most important of which are sparse matrix vector product (SpMV) and 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. 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 [3].

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 will review the sparse matrix storage formats used in this work and in Section 5 we will explain how we have implemented the basic operations used in order to perform the algorithms described in Sections 1 and 2. Finally, in Section 6, we display the numerical results obtained using CUDA over a GPU and we compare these results with those obtained on the shared memory platform using an OpenMP model. Furthermore, a mixed model is considered in order to exploit the characteristics of both parallel systems.

## 2 Nonlinear Preconditioned CG Method

Preconditioning is a technique for improving the condition number ($\mathrm{cond}$) 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 $\mathrm{cond}(M^{-1}A) << \mathrm{cond}(A)$ 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 $i = 0, 1, \ldots,$ until convergence
    $\alpha_i \Longrightarrow$ 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 [4]. These preconditioners consist of considering a splitting of the matrix $A$ as

$$A = P - Q \tag{3}$$

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 $s = (I + R + R^2 + \ldots + R^{m-1})P^{-1}r$, where $R = P^{-1}Q$ and the preconditioning matrix is $M_m = P(I + R + R^2 + \ldots + R^{m-1})^{-1}$, cf. [4].
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

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \Phi_1(x) \\ \Phi_2(x) \\ \vdots \\ \Phi_p(x) \end{bmatrix}, \tag{4}$$

where $x$ and $\Phi(x)$ are partitioned according to the size of the blocks of $A$. Let us consider the splitting (3) such that $P$ consists of the diagonal blocks of $A$ in (4), that is

$$P = \operatorname{diag}(A_{11}, \ldots, A_{pp}). \tag{5}$$

Note that in this case, performing $m$ steps of the iterative procedure defined by the splitting (3) to approximate the solution of $As = r$, corresponds to perform $m$ steps of the Block Jacobi method. Thus, at each step $l$, $l = 1, 2, \ldots,$ of a Block Jacobi method, $p$ independent linear systems of the form

$$A_{jj} s_j^{(l)} = (Qs^{(l-1)} + r)_j, \;\; 1 \le j \le p, \tag{6}$$

need to be solved; therefore each linear system (6) can be solved by a different processor. However, when the order of the diagonal blocks $A_{jj}$, $1 \leq j \leq p$, is large, it is natural to approximate their solutions by using an iterative method, and thus we are in the presence of a two-stage iterative method; see e.g., [2]. In a formal way, let us consider the splittings

$$A_{jj} = B_j - C_j, \ \ 1 \leq j \leq p, \tag{7}$$

and at each $l$th step perform, for each $j$, $1 \leq j \leq p$, $q(j)$ iterations of the iterative procedure defined by the splittings (7) to approximate the solution of (6). That is, to solve the auxiliary system $Ms = r$ of Algorithm 2, we use $m$ steps of the iteration $s^{(l)} = Ts^{(l-1)} + W^{-1}r$, $\ l = 1, 2, \ldots, m$, choosing $s^{(0)} = 0$, where

$$T = H + (I - H)P^{-1}Q, \ \ W = P(I - H)^{-1}, \tag{8}$$

with $P$ defined in (5) and $H = \mathrm{diag}((B_1^{-1}C_1)^{q(1)}, \ldots, (B_p^{-1}C_p)^{q(p)})$; see e.g., [5]. This linear method is established in the following algorithm [2].

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

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

We note that the updated vector from $m$ steps of Algorithm 3 with $s^{(0)} = 0$ is given by $s^{(m)} = (I + T + T^2 + \ldots + T^{m-1})W^{-1}r$ where $T$ and $W$ are defined in (8). Therefore, the preconditioner related to the block two-stage method is given by $M_m = W(I + T + T^2 + \ldots + T^{m-1})^{-1}$.

## 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 [6, 7], an application consists of a sequential host program, that may execute parallel programs known as kernels on a parallel device, i.e. GPU. Note 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 owned by a thread. The constant and the texture memories have on-chip cache, none of them will be used due to the characteristics of our problem.

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 30K threads. In order to manage efficiently this large number of threads, the GPU employs an SIMT (Single Instruction Multiple Thread) architecture [6, 8] 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

In this section, we describe the sparse matrix formats used in order to optimize the sparse matrix vector product. Concretely, we have used the Compressed Row Storage (CSR) format, the ELLPACK (or ITPACK) format [9], and the ELLPACK-R format proposed in [10]. There are a multitude of sparse matrix representations, each one with different storage requirements, computational characteristics, and methods of accessing and manipulating entries of the matrix. In the context of sparse matrix vector product on GPU, we have only considered common sparse matrix formats suitable to obtain a good behavior for computing on the GPU.

The compressed row storage (CRS) format is a popular, general-purpose sparse matrix representation, it makes absolutely no assumptions about the sparsity structure of the matrix, and it does not store any unnecessary element. The CRS format puts the subsequent nonzero elements of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix $A$, we create three vectors: one for floating point numbers and other two for integers. The floating point vector stores the values of the nonzero row 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, therefore the last entry corresponds to NNZ (NNZ+1 if the first element is equal to 1 instead of 0), the number of nonzero elements in the matrix.

ELLPACK [9] 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. Both arrays are of dimension at least $N * MaxEntriesbyRows$, where $N$ is the number of rows and $MaxEntriesbyRows$ is the maximum number of nonzero elements per row in the matrix. Note that the size of all rows in these compressed arrays is the same, for this purpose every row is padded with zeros. Therefore, ELLPACK can be considered as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. Consequently, this format, as we said above, is appropriate to compute operations with sparse matrices on vector architectures. However, 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.

ELLPACK-R is a variant of the ELLPACK format introduced in [10] with the purpose to optimize the sparse matrix vector product in GPUs. ELLPACK-R consists of two arrays of dimension $N * MaxEntriesbyRows$, following original ELLPACK; moreover, an additional integer array of dimension $N$ is included with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded. Note that the arrays must store their elements in column-major order. The advantage of ELLPACK-R (see [10]) are: the coalesced global memory access, thanks to the column-major ordering used to store the matrix elements into the data structures; as the two previous formats, it is well-suited for computing sparse matrix vector product on GPU because it allows non-synchronized execution between different blocks of threads; the reduction of the waiting time or unbalance between threads of one warp; and homogeneous computing within the threads in the warps.

## 5 Basic Operations

Following Algorithm 1 and Algorithm 2, the basic operations to implement the methods proposed are:

- Sparse matrix vector product.
- Vector operations (included in level 1 BLAS [11]).
- Inner product.
- LU solver (included in SPARSKIT [12] only for NLPCG method; see Section 6).

In the rest of this section, we describe the 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.

## 5.1 Sparse Matrix Vector Product

In order to compute the sparse matrix vector product (SpMV), we must consider the different sparse matrix formats described in Section 4. Particularly, the kernel code to compute the SpMV, using CSR 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 optimize the access to GPU global memory modifying the thread mapping. However, in the second case the optimizations performed are focused on matrices with higher sparsity pattern than the matrices of the test example used in our work (see Section 6). Note that, in our test example the typical number of nonzero elements by row is 7, and, for example, the optimizations presented in [13] need more than 32 nonzero elements per row. Consequently, in order to optimize the SpMV operation, we will consider the use of ELLPACK and ELLPACK-R storage formats. Note that in both formats the memory access pattern is improved.

On the other hand, in order to compute the SpMV we use CUSPARSE [14]. 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 CSR format.

## 5.2 Vector Operations

As we can see in Algorithm 1 and Algorithm 2, the common vector operations, without 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. On the other hand, CUBLAS[15] 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 *VectorN* vectors of *ElementN* elements. *ElementN* 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 accumulator. *ACCUM_N* must be a power of two and preferably 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. Note that CPU must complete the operation by computing the *VectorN* partial results. The kernel code for computing a single inner product is the following, but in our code we group several inner products in only one kernel.

CUDA Inner Product

```
__global__ void Inner_Prodcut(double* resprod ,double* x,
                         double* y,int vectorN ,int elementN){
    __shared__ double accumResult[ACCUM_N];
    for(int vec=blockIdx.x;vec<vectorN;vec+=gridDim.x){
        int vectorBase = IMUL(elementN ,vec);
        int vectorEnd = vectorBase+elementN;
        for(int iAccum=threadIdx.x;iAccum<ACCUM_N;iAccum+=blockDim.x){
            double sum = 0;
            for(int pos=vectorBase+iAccum;pos<vectorEnd;pos+=ACCUM\_N){
                sum += x[pos] * y[pos];}
             accumResult[iAccum] = sum;}
        for(int stride = ACCUM\_N / 2; stride > 0; stride >>= 1){
            __syncthreads();
            for(int iAccum=threadIdx.x;iAccum<stride;iAccum+=blockDim.x){
                accumResult[iAccum]+=accumResult[stride+iAccum];}}
        if(threadIdx.x==0) {
            resprod[vec] = accumResult[0];}}
}
```

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 fine grain inherent parallelism. We have tried various strategies to compute the LU solver on the GPU but efficiency has not been achieved. Moreover the technique used in order to exploit multicore CPU architectures becomes ineffective due to CPU-GPU communications. Due to the structure of (4) and assuming that $p$ is the number of cores, the speed of convergence of the NLPCG method in a multicore decreases when the number of cores increases; 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 arrays to complete reduction operations.

## 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

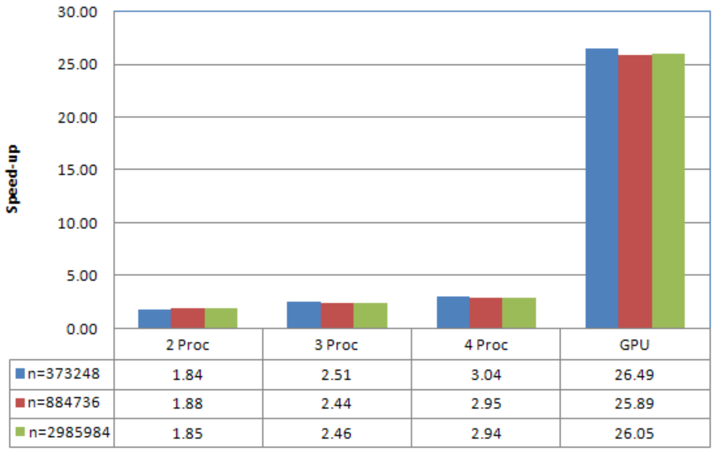
$$\nabla^2 u - \lambda e^u = 0, \tag{9}$$

where $u$ is the temperature and $\lambda$ is a constant known as the Frank-Kamenetskii parameter; see e.g., [3]. There are two possible steady-state solutions to this problem for a given value of $\lambda$. 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 $\Omega$ of unit length and $\lambda = 6$. To solve equation (9) using the finite difference method, we consider a grid in $\Omega$ 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 $i$th component $\Phi_i$ of $\Phi$ is a function only of the $i$th 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 analysis performed are based on the run-times measured on a GeForce GTX 280 compared with the parallel run-times measured on SULLI using OpenMP. First we present results for the NLCG method with problems of various sizes. In Figure 1 we show the speed-up using OpenMP and different number of cores in SULLI, and the speed-up when the NLCG method is computed in the NVIDIA GeForce GTX 280 GPU, managed from one core of SULLI. We obtain a good speed-up with OpenMP using the available cores, but not comparable with the speed-up obtained with GPU, greater than 25. These results confirm a good interaction between the NLCG algorithm and a GPU computing platform.

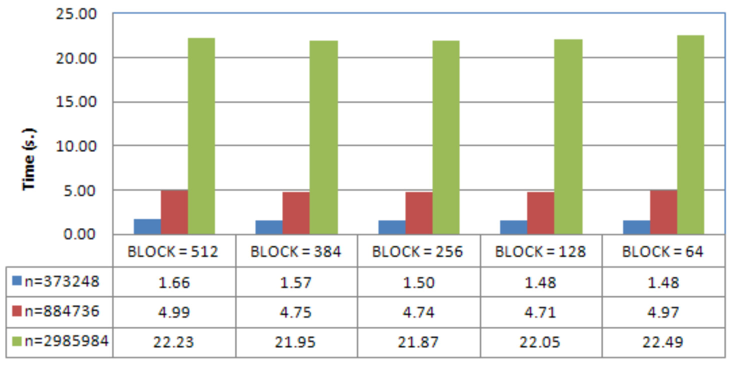
In Figure 2 we analyze the behavior of the NLCG algorithm with respect to the number of threads in each block. Note that, in order to call a CUDA kernel, we set the number of threads of each block and the maximum number of threads is $512$. Therefore, the number of blocks in each grid depends on the number of threads required. For example, if each block has 512 threads, for calling the *axpy* kernel working with a system of size $n = 373248$, it is required a grid with $729$ blocks. 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. In Figure 2 we present results varying the block size, with *ACCUM_N* equal to $128$; note that we work grouping operations and therefore we work with several arrays in the GPU shared memory. The best performance is obtained using $256$ or $128$ threads in each block. These results can be extended to all experiments performed.

In Figure 3 we analyze the influence of *ACCUM_N*, that is, the size of the shared memory arrays that act as accumulators. Remark that we can not work with high sizes



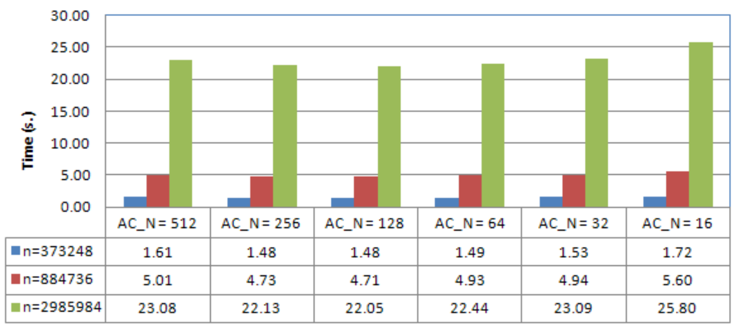
| | 2 Proc | 3 Proc | 4 Proc | GPU |
|---|---|---|---|---|
| 🟦 n=373248 | 1.84 | 2.51 | 3.04 | 26.49 |
| 🟥 n=884736 | 1.88 | 2.44 | 2.95 | 25.89 |
| 🟩 n=2985984 | 1.85 | 2.46 | 2.94 | 26.05 |

Figure 1: Speed-up NLCG method.



| | BLOCK = 512 | BLOCK = 384 | BLOCK = 256 | BLOCK = 128 | BLOCK = 64 |
|---|---|---|---|---|---|
| 🟦 n=373248 | 1.66 | 1.57 | 1.50 | 1.48 | 1.48 |
| 🟥 n=884736 | 4.99 | 4.75 | 4.74 | 4.71 | 4.97 |
| 🟩 n=2985984 | 22.23 | 21.95 | 21.87 | 22.05 | 22.49 |

Figure 2: NLCG method, ACCUM_N= $128$.



| | AC_N= 512 | AC_N= 256 | AC_N= 128 | AC_N= 64 | AC_N= 32 | AC_N= 16 |
|---|---|---|---|---|---|---|
| 🟦 n=373248 | 1.61 | 1.48 | 1.48 | 1.49 | 1.53 | 1.72 |
| 🟥 n=884736 | 5.01 | 4.73 | 4.71 | 4.93 | 4.94 | 5.60 |
| 🟩 n=2985984 | 23.08 | 22.13 | 22.05 | 22.44 | 23.09 | 25.80 |

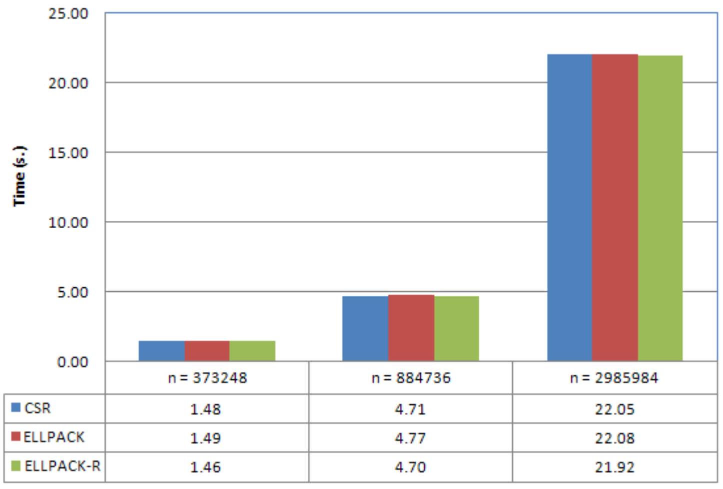
Figure 3: NLCG method, BLOCK = 128.

because we declare more than one array in each kernel that implements reduction operation. Figure 3 presents results calling kernels with 128 threads per block and for different system sizes. As it can be seen, though the best value to be selected is 128, the influence of *ACCUM_N* is not critical. In this case, it is also possible to extend this conclusion to all experiments performed.

In Figure 4 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. The algorithm for computing SpMV using ELLPACK-R does not include flow control instructions that serialize the execution of a warp of 32 threads and it allows coalesced matrix data access. Note that ELLPACK-R format is the sparse matrix storage format with highest memory requirement from among the formats discussed. This is due to using a vector for 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. In [10] it is confirmed that the performance improvement for matrices with high sparsity pattern is not significant. The results shown in Figure 4 have been obtained using optimal values for the rest of parameters.

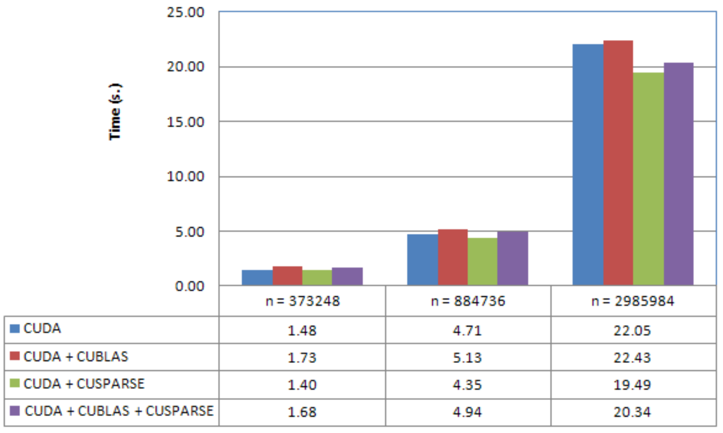
To conclude the analysis of the NLCG method , we present in Figure 5 results using the CUBLAS and CUSPARSE libraries, included in the CUDA Toolkit 3.2 RC. We analyze the use of CUBLAS and the use of CUSPARSE separately, and the use of both together. First, when we only use CUBLAS library the results are worse than the best results using only the CUDA API. 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. Remark that, CUBLAS and CUSPARSE libraries hide to the user the setting of parameters, both for calling kernels as in the reduction operations.

In order to analyze the NLPCG method (see Section 2), we consider the outer splitting $A = P - Q$ determined by $P = \mathrm{diag}(A_{11}, \ldots, A_{pp})$. Note that, as we have mentioned in Section 5.4, the LU solver is computed in the multicore, therefore $p$ is the number of cores. Let us further consider an incomplete LU factorization of



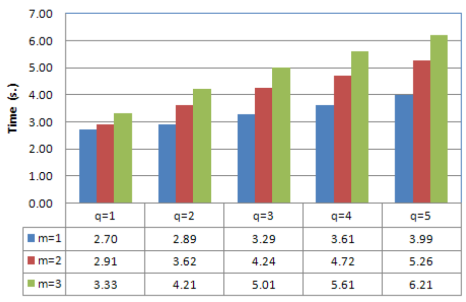
| | n = 373248 | n = 884736 | n = 2985984 |
|---|---|---|---|
| 🟦 CSR | 1.48 | 4.71 | 22.05 |
| 🟥 ELLPACK | 1.49 | 4.77 | 22.08 |
| 🟩 ELLPACK-R | 1.46 | 4.70 | 21.92 |

Figure 4: NLCG method vs sparse matrix storage format.



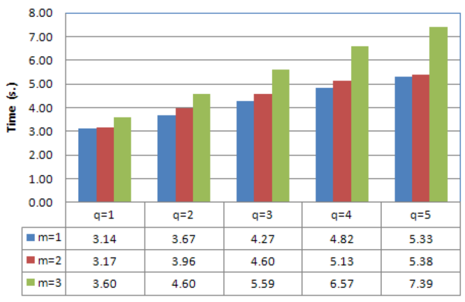
| | n = 373248 | n = 884736 | n = 2985984 |
|---|---|---|---|
| 🟦 CUDA | 1.48 | 4.71 | 22.05 |
| 🟥 CUDA + CUBLAS | 1.73 | 5.13 | 22.43 |
| 🟩 CUDA + CUSPARSE | 1.40 | 4.35 | 19.49 |
| 🟪 CUDA + CUBLAS + CUSPARSE | 1.68 | 4.94 | 20.34 |

Figure 5: NLCG method using CUBLAS and/or CUSPARSE.



| | 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 |

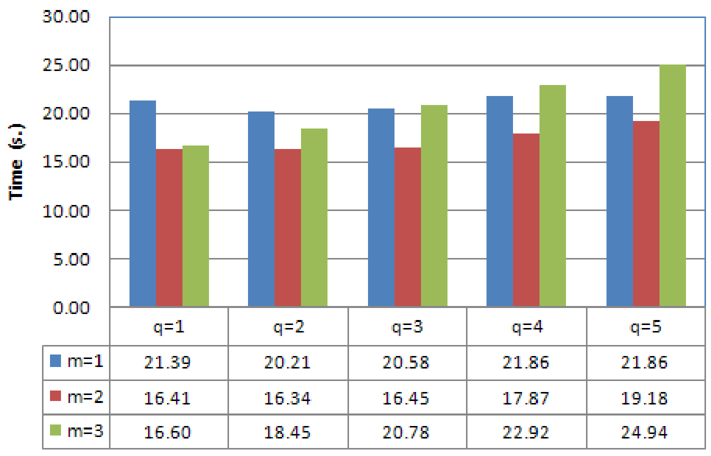
(a) ILU(0)



| | 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)

Figure 6: NLPCG, 1 core + GPU, $n = 373248$.



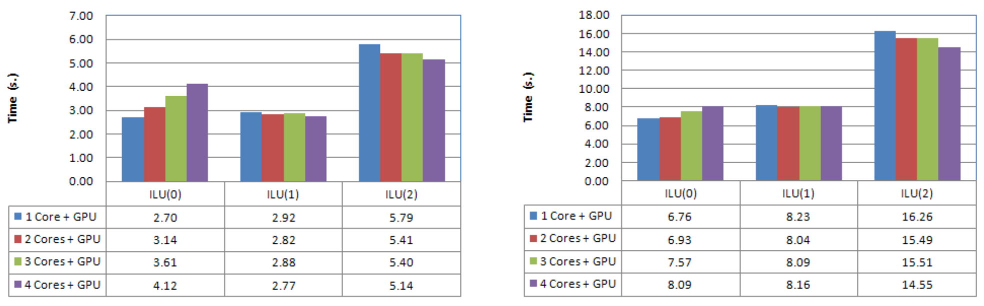
| | q=1 | q=2 | q=3 | q=4 | q=5 |
|---|---|---|---|---|---|
| 🟦 m=1 | 21.39 | 20.21 | 20.58 | 21.86 | 21.86 |
| 🟥 m=2 | 16.41 | 16.34 | 16.45 | 17.87 | 19.18 |
| 🟩 m=3 | 16.60 | 18.45 | 20.78 | 22.92 | 24.94 |

Figure 7: NLPCG, 2 cores + GPU, $n = 884736$, ILU(0).

each matrix $A_{jj}$, $j = 1, 2, \ldots, p$, that is $A_{jj} = L_jU_j - R_j$, 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 [16], ILU$(\kappa)$, $\kappa \geq 0$, 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 $\kappa$ are removed from $S$. In the experiments reported here, we have used these ILU$(\kappa)$ factorizations for the matrices $A_{jj}$, $j = 1, 2, \ldots, p$, defined above.

First, we present results showing the behavior of the characteristic parameters of the NLPCG method, 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. Initially, we consider only one block, thus the method uses one core and the GPU. In Figure 6 we present results varying the number of inner iterations $q$ and the number of outer iterations $m$, for a system of size equal to $373248$ and level of fill-in equal to $0$ and equal to $1$. Figure 6 shows that the best results are obtained using $q = 1$ and $m = 1$. The experiments performed show analogous results to the results obtained in a multicore architecture, where the best results were obtained with small values of $q$ and $m$ (see [17]).

We present results using $2$ cores and the GPU to solve a system of size $884736$ in Figure 7. The results correspond to a level of fill-in equal to $0$. In this case the optimal value of the number of outer iterations is $m = 2$. Since the LU solver is performed at the CPU, using more than one core can benefit the algorithm; however, it is important to remark that using the optimal values it should not be used more than one core, because the increase in the number of global iterations required does not allow an improvement. This performance using optimal values is presented in Figure 8.

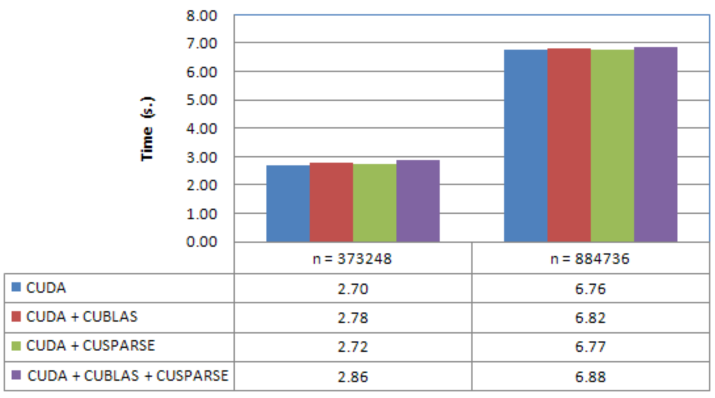


| | ILU(0) | ILU(1) | ILU(2) |
|---|---|---|---|
| 🟦 1 Core + GPU | 2.70 | 2.92 | 5.79 |
| 🟥 2 Cores + GPU | 3.14 | 2.82 | 5.41 |
| 🟩 3 Cores + GPU | 3.61 | 2.88 | 5.40 |
| 🟪 4 Cores + GPU | 4.12 | 2.77 | 5.14 |

| | ILU(0) | ILU(1) | ILU(2) |
|---|---|---|---|
| 🟦 1 Core + GPU | 6.76 | 8.23 | 16.26 |
| 🟥 2 Cores + GPU | 6.93 | 8.04 | 15.49 |
| 🟩 3 Cores + GPU | 7.57 | 8.09 | 15.51 |
| 🟪 4 Cores + GPU | 8.09 | 8.16 | 14.55 |

(a) $n = 373248$ (b) $n = 884736$

Figure 8: NLPCG, $m = 1$, $q = 1$.



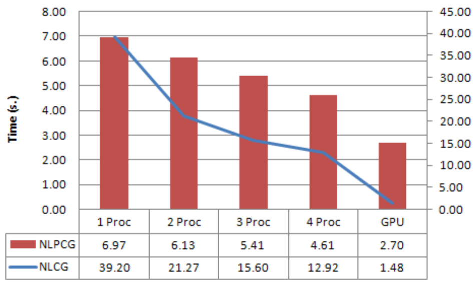
| | n = 373248 | n = 884736 |
|---|---|---|
| 🟦 CUDA | 2.70 | 6.76 |
| 🟥 CUDA + CUBLAS | 2.78 | 6.82 |
| 🟩 CUDA + CUSPARSE | 2.72 | 6.77 |
| 🟪 CUDA + CUBLAS + CUSPARSE | 2.86 | 6.88 |

Figure 9: NLPCG, 1 core + GPU, $m = 1$, $q = 1$, ILU(0).

The conclusions about the optimal value of the number of threads in each block have been analogous to those obtained in the NLCG method. That is, the optimal block size is $128$ or $256$, but the choice of a non-optimal block size is not critical. On the other hand, the behavior of the NLPCG algorithm is not affected by the value of *ACUMM_N*.

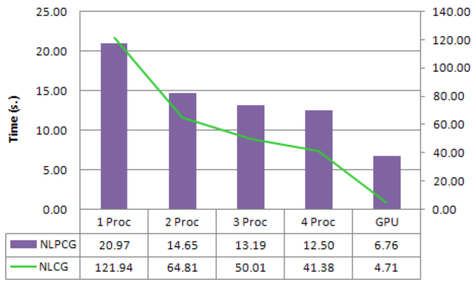
Figure 9 shows that the use of CUBLAS in the NLPCG method slightly increases the computational times, as we have also seen in the NLCG method. Moreover, the NLPCG method is not affected by using CUSPARSE. It is due to the small number of iterations of the NLPCG method, which is nearly $10$ times lower than the iterations needed by the NLCG method .

Finally, from the analysis of Figure 10 it deduces that the speed-up using GPU is greater than the speed-up using four cores of the CPU. Nevertheless the speed-up is not comparable with the value obtained by the NLCG method, which was above 25. 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. Clearly, as we have explained, both methods present very different inherent parallelism, therefore their performance on two different parallel architectures is also



| | 1 Proc | 2 Proc | 3 Proc | 4 Proc | GPU |
|---|---|---|---|---|---|
| 🟥 NLPCG | 6.97 | 6.13 | 5.41 | 4.61 | 2.70 |
| — NLCG | 39.20 | 21.27 | 15.60 | 12.92 | 1.48 |

(a) $n = 373248$



| | 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 | 4.71 |

(b) $n = 884736$

Figure 10: Comparison NLCG and NLPCG on CPU and GPU.

different.

## 7 Conclusion

We have developed the Fletcher-Reeves version of the nonlinear conjugate gradient method and we have applied a polynomial preconditioner type based on the two-stage methods, using the GPGPU technique (General-purpose computing on graphics processing units). 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 the most efficient format. On the other hand, we have shown differences in adaptation of both methods to fine grain GPU architecture. We would like to point out that the use of the GPU improves the results obtained using any of the proposed methods and on the other hand the NLCG method exploits better the parallelism offered by the GPU than the NLPCG method.

## Acknowledgements

This research was supported by the Spanish Ministry of Science and Innovation under grant number TIN2008-06570-C04-04.

## References

[1] R. Fletcher, C. Reeves, “Function Minimization by Conjugate Gradients”, The Computer Journal, 7, 149-154, 1964.

[2] R. Bru, V. Migallón, J. Penadés, D.B. Szyld, "Parallel, Synchronous and Asynchronous Two-Stage Multisplitting Methods", Electronic Transactions on Numerical Analysis, 3, 24-38, 1995.

[3] 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.

[4] L. Adams, "M-step preconditioned conjugate gradient methods", SIAM Journal on Scientific and Statistical Computing, 6, 452-462, 1985.

[5] V. Migallón, J. Penadés, "Convergence of two-stage iterative methods for hermitian positive definite matrices", Applied Mathematics Letters, 10(3), 79-83, 1997.

[6] J. Nickolls, I. Buck, M. Garland, K. Skadron, "Scalable parallel programming with CUDA", Queue, 6(2), 40-53, 2008.

[7] 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`

[8] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture", IEEE Micro, 28(2), 39-55, 2008.

[9] D.R. Kincaid, D.M. Young, "A brief review of the ITPACK Project", Journal of Computational and Applied Mathematics, 24(1-2), 121-127, 1988.

[10] F. Vázquez, J.J. Fernández, E.M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs", Concurrency and Computation: Practice and experience, DOI: 10.1002/cpe.1658, 2010.

[11] C.L. Lawson, R.J. Hanson, D. Kincaid, F.T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage", ACM Transactions on Mathematical Software, 5, 308-323, 1979.

[12] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computation", `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[13] M. Manikandan, R. Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs", IBM Research Report RC24704, 2008.

[14] 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`.

[15] 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`.

[16] H.P. Langtangen, "Conjugate gradient methods and ILU preconditioning of nonsymmetric matrix systems with arbitrary sparsity patterns", International Journal for Numerical Methods in Fluids, 9, 213-233, 1989.

[17] H. Migallón, V. Migallón, J. Penadés, "Parallel Nonlinear Conjugate Gradient Algorithms on Multicore Architectures", Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering, 689-700, 2009.