Journal of Computational and Applied Mathematics 236 (2012) 3584–3590

Contents lists available at SciVerse ScienceDirect

Journal of Computational and Applied Mathematics

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

Parallel preconditioned conjugate gradient algorithm on GPU

Rudi Helfenstein, Jonas Koko *

Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 Clermont-Ferrand, France
CNRS, UMR 6158, LIMOS, F-63173 Aubière, France

ARTICLE INFO

Article history:
Received 30 September 2010
Received in revised form 1 April 2011

Keywords:
Preconditioned conjugate gradient
Parallel computing
Graphics processor unit

ABSTRACT

We propose a parallel implementation of the Preconditioned Conjugate Gradient algorithm on a GPU platform. The preconditioning matrix is an approximate inverse derived from the SSOR preconditioner. Used through sparse matrix–vector multiplication, the proposed preconditioner is well suited for the massively parallel GPU architecture. As compared to CPU implementation of the conjugate gradient algorithm, our GPU preconditioned conjugate gradient implementation is up to 10 times faster (8 times faster at worst).

1. Introduction

In the last years, *Graphics Processing Units* (GPU) have evolved into a very flexible and powerful many-core processor. Indeed, the modern GPUs are specialized for compute-intensive, massively parallel computation, e.g., rendering real-time realistic 3D environment. The fast-growing video game industry exerts strong economic pressure that forces constant innovation. The massively parallel architecture offers tremendous performance in many high-performance computing applications. Numerical algorithms can be significantly accelerated if the algorithms map well to the characteristic of the GPU.

In this paper, we focus on the numerical solution of the generalized Poisson equation. The Poisson equation arises in many applications in computational fluid dynamics, electrostatics, magnetostatics, image processing, etc. Numerical solution of the Poisson equation, through finite element or finite difference discretization, leads to large sparse linear systems usually solved by iterative methods instead of direct methods (Gaussian elimination or Cholesky factorization).

The conjugate gradient (CG) algorithm is one of the best known iterative methods for solving linear systems with symmetric, positive definite matrix. The method is easy to implement and, for the Poisson equation, can handle complex domains and boundary conditions. The CG method can be easily adapted for linear systems with symmetric, semi-definite positive matrix (see, e.g., [1]). With a suitable preconditioner, the performance can be dramatically increased. The preconditioned conjugate gradient (PCG) has proven its efficiency and robustness in a wide range of applications.

Preconditioning consists of replacing the original linear system by one which as the same solution, but which is likely to be easier to solve with an iterative solver. Our goal is to develop a suitable and flexible PCG algorithm for the GPU architecture. Standard preconditioning techniques like incomplete factorizations or Symmetric Successive Over-Relaxation (SSOR) are hard to parallelize because of their strongly serial processing due to the forward/backward substitutions. Simple preconditioners like Jacobi has a limited impact on the efficiency and robustness of the method. The approximate inverse preconditioners have attractive features for GPU. First, the columns or rows of the approximate inverse matrix can be generated in parallel. Second, the preconditioner matrix is used in PCG through matrix–vector multiplications, easier to parallelize than forward/backward substitutions. But the approximate inverse techniques suffer from lack of robustness. The fully parallel technique does not guarantee that the resulting approximate inverse is neither symmetric nor positive definite.

* Corresponding author at: Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 Clermont-Ferrand, France. Tel.: +33 4 73 40 50 01.
E-mail address: koko@sp.isima.fr (J. Koko).

0377-0427/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2011.04.025

The efficient preconditioning has been largely ignored in previous work, except in [2] where a heuristic approximate inverse, based on SSOR preconditioner, is proposed as preconditioner for the PCG algorithm. The approximate inverse is derived on a rectangular domain with a regular grid (finite difference method). Our approximate SSOR inverse is derived rigorously using a first order approximation and is independent of the discretization method (finite element or finite difference method). The PCG algorithm presented in this paper can then be used for any linear system with positive definite matrix (not necessarily arisen from discretization of the Poisson equation).

The paper is organized as follows. In Section 2, we present the Preconditioned Conjugate Gradient algorithm. The derivation of the SSOR approximate inverse preconditioner is described in Section 3. The implementation of the algorithm on GPU is presented in Section 4, followed by numerical experiments in Section 5.

2. Conjugate gradient algorithm

2.1. Motivation

Let Ω be a bounded, open domain in $\mathbb{R}^d$, $d = 2, 3$. The generalized Poisson equation is

$$\alpha u - \nu \Delta u = f, \quad \text{in } \Omega, \tag{2.1}$$

where $\alpha \geq 0$ and $\nu > 0$. A solution u of (2.1) must satisfy boundary conditions (Dirichlet, Neumann or Cauchy). Eq. (2.1) arises in a wide range of physical problems (computational fluid dynamics, magnetostatics, electrostatics, electronic device simulation, etc.).

A discretization of (2.1) (using finite element or finite difference) leads to the linear system

$$Ax = b, \tag{2.2}$$

where A is real, symmetric and positive definite. Many solution methods exist:

- direct methods (Gaussian elimination, Cholesky decomposition), [3–5]
- iterative methods (Jacobi, conjugate gradient, etc.), [3,6,7,5,8].

In general, the matrix A is large and sparse so that the direct methods become impracticable.

2.2. Conjugate gradient

The conjugate gradient (CG) is one of the best known iterative methods for solving sparse symmetric positive definite linear systems. The method is flexible, easy to implement and converges (theoretically) in a finite number of steps. Furthermore, its implementation requires only matrix–vector multiplications. The conjugate gradient algorithm is as follows.

Conjugate Gradient (CG)

$k = 0$: Initialization: x_0, $p_0 = r_0 = b - Ax_0$
$k \geq 0$: while $\|r_k\|/\|r_0\| > \varepsilon$

1. $q_k = Ap_k$, $\alpha_k = \frac{\|r_k\|^2}{p_k^T q_k}$
2. $x_{k+1} = x_k + \alpha_k p_k$, $r_{k+1} = r_k - \alpha_k q_k$
3. $\beta_k = \frac{\|r_{k+1}\|^2}{\|r_k\|^2}$, $p_{k+1} = r_{k+1} + \beta_k p_k$.

Each iteration requires one matrix–vector product and two inner products. All the necessary operations can be found in a standard library (e.g. BLAS). In addition to the matrix A and the approximate solution x, we have to store three auxiliary vectors (r, p and q).

Define the A-inner product by

$$(x, y)_A = x^T Ay$$

and the corresponding A-norm

$$\|x\|_A = \sqrt{x^T Ax}.$$

A speed of convergence can be given in terms of A-norm and condition number $\kappa = \kappa_2(A) = \lambda_{\max}/\lambda_{\min}$, where $\lambda_{\max}$ and $\lambda_{\min}$ are the greatest and the lowest eigenvalue of A, respectively. If x^* is the solution of (2.2), then the sequence $\{x_k\}$ generated by Algorithm CG is such that (see, e.g., [7,8])

$$\|x^* - x_k\|_A \leq 2\|x^* - x_0\|_A \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)^k.$$

A simple heuristic rule for the fast convergence of the conjugate gradient algorithm is $\kappa_2(A) \approx 1$. But in industrial applications, like computational fluid dynamics or electronic device simulation, the matrix A can have a large condition number, i.e. $\kappa_2(A) \gg 1$. The efficiency of the CG algorithm can be significantly improved by *preconditioning*.

The idea behind preconditioning is to replace (2.2) by

$$M^{-1}Ax = M^{-1}b \tag{2.3}$$

or

$$AM^{-1}y = b, \quad x = M^{-1}y, \tag{2.4}$$

where M is also symmetric positive definite. Eq. (2.3) corresponds to a *left preconditioner* whereas (2.4) corresponds to a *right preconditioner*. The matrix M must be such that $\kappa_2(M^{-1}A) \ll \kappa_2(A)$ or $\kappa_2(AM^{-1}) \ll \kappa_2(A)$, and $Mz = r$ is inexpensive to solve. The preconditioned version of Algorithm CG is as follows.

Preconditioned Conjugate Gradient (PCG)

$k = 0$: Initialization: x_0, $r_0 = b - Ax_0$, $Mz_0 = r_0$, $p_0 = z_0$
$k \geq 0$: while $\|r_k\|/\|r_0\| > \varepsilon$

1. $q_k = Ap_k$, $\alpha_k = \frac{z_k^T r_k}{p_k^T q_k}$
2. $x_{k+1} = x_k + \alpha_k p_k$, $r_{k+1} = r_k - \alpha_k q_k$
3. $Mz_{k+1} = r_{k+1}$
4. $\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$, $p_{k+1} = r_{k+1} + \beta_k p_k$.

The additional cost is one linear system per iteration (to compute z_{k+1}). This sequence of computations is valid for both right and left preconditioners. The left preconditioned CG algorithm with M-inner product is mathematically equivalent to the right preconditioned CG algorithm with M^{-1}-inner product.

3. SSOR preconditioner

For the left preconditioner (2.4), one of the simplest ways is to perform an incomplete (LU or Cholesky) factorization. This incomplete factorization is rather easy and inexpensive to implement. The linear system in Step 3 of Algorithm PCG then reduces to forward/backward substitutions. But this leads to strongly serial loops, not suitable for modern GPUs.

To avoid solving linear systems, one possible way is to try to find a preconditioner that does not require solving a linear system. This can be done by computing M as a direct approximation to the inverse of A (see, e.g., [9,10,8]). This problem can be formulated as the following minimization problem

$$\min_M F(M) = \frac{1}{2}\|I - AM\|_F^2 \tag{3.1}$$

for the right-approximate inverse; see, e.g., [9,10]. The main disadvantage of the Frobenius norm minimization approach is that it is difficult to predict whether the resulting approximate inverse is non-singular. Theoretically, it cannot be proved that the approximate inverse M obtained by (3.1) is non-singular unless the approximation is accurate enough. But one of the requirement, for a "good" preconditioner, is to keep M sparse.

An approximate (and sparse) inverse can be obtained easily using *Symmetric Successive Over-Relaxation* (SSOR). Assume that the matrix A is decomposed as follows

$$A = L + D + L^T,$$

where D is the diagonal matrix of diagonal elements of A and L the lower triangular part of A. The SSOR preconditioner is defined by

$$M = KK^T, \tag{3.2}$$

where

$$K = \frac{1}{\sqrt{2-\omega}}(\bar{D} + L)\bar{D}^{-1/2}, \tag{3.3}$$

where $0 < \omega < 2$ and $\bar{D} = (1/\omega)D$. As noted in the previous section, the factorized form (3.2) is not suitable for GPU. In contrast with the previous preconditioning techniques, we can compute an approximate inverse of K explicitly. The factor K can be rewritten as

$$K = \frac{1}{\sqrt{2-\omega}}\bar{D}(I + \bar{D}^{-1}L)\bar{D}^{-1/2}$$

such that

$$K^{-1} = \sqrt{2-\omega}\,\bar{D}^{1/2}(I + \bar{D}^{-1}L)^{-1}\bar{D}^{-1}.$$

Define $\rho(A)$, the spectral radius of a matrix A. Assuming that $\rho(\bar{D}^{-1}L) < 1$, then a Neumann series approximation of the inverse of K is

$$K^{-1} \approx \sqrt{2-\omega}\bar{D}^{1/2}\left[I - \bar{D}^{-1}L + (\bar{D}^{-1}L)^2 - (\bar{D}^{-1}L)^3 + \cdots\right]\bar{D}^{-1}. \tag{3.4}$$

A first order approximate inverse of K is given by

$$K^{-1} \approx \sqrt{2-\omega}\bar{D}^{1/2}(I - \bar{D}^{-1}L)\bar{D}^{-1} = \sqrt{2-\omega}\bar{D}^{-1/2}(I - L\bar{D}^{-1}) =: \bar{K}. \tag{3.5}$$

$\bar{K}$ can be computed easily by using A and reciprocal operations in each diagonal element of A. Moreover, $\bar{K}$ has the same sparsity pattern as A. Indeed,

$$\bar{K}_{ij} = \sqrt{2-\omega}\left(\frac{\omega}{a_{ii}}\right)^{1/2}\left(\delta_{ij} - \omega\frac{a_{ij}}{a_{jj}}\right), \quad j \le i.$$

The SSOR approximate inverse (SSOR-AI) preconditioner is therefore

$$\bar{M} = \bar{K}^T\bar{K}$$

and Step 3 in Algorithm PCG is replaced by $z_{k+1} = \bar{M}r_{k+1}$. Note that $\bar{M}$ can be computed with a prescribed sparsity pattern (e.g. the one of A) to reduce the computational cost.

In [2], Ament et al. [2] proposed the following approximate inverse preconditioner (obtained by using a heuristic approach)

$$\tilde{M} = (I - LD^{-1})(I - D^{-1}L). \tag{3.6}$$

They called it incomplete Poisson (IP) preconditioner. Note that for $\omega = 1$, (3.5) is

$$\bar{K} = D^{-1/2}(I - LD^{-1})$$

such that

$$\tilde{M} = D^{1/2}\bar{K}\bar{K}^T D^{1/2}.$$

4. GPU implementation

4.1. Matrix storage

To take advantage of the large number of zeros in matrices issue from the discretization of PDEs, special storage formats are required. The main idea is to keep only non-zero elements, while allowing common matrix operations. For our numerical experiments, we adopt the *Compressed Sparse Row* (CSR) format (see e.g. [11,8]). In CSR format, an $n \times n$ sparse matrix A, with nnz non-zero elements, is stored via three arrays:

`AA` (array of length nnz) contains the non-zero entries of A, stored row by row;
`JA` (array of length nnz) contains the column indices of the non-zero entries stored in `AA`;
`IA` (array of length $n + 1$) contains the pointers (indices) to the beginning of each row in the arrays `AA` and `JA`.

The following matrix

$$A = \begin{bmatrix} 2 & 0 & 1 & 3 & 0 \\ 0 & 4 & 0 & 0 & 1 \\ 1 & 0 & 0 & 2 & 4 \\ 0 & 1 & 1 & 2 & 4 \\ 5 & 0 & 1 & 0 & 3 \end{bmatrix}$$

is stored in CSR format by

`IA` :	1	4	6	9	13	16

`JA` :	1	3	4	2	5	1	4	5	2	3	4	5	1	3	5

`AA` :	2	1	3	4	1	1	2	4	1	1	2	4	5	1	3

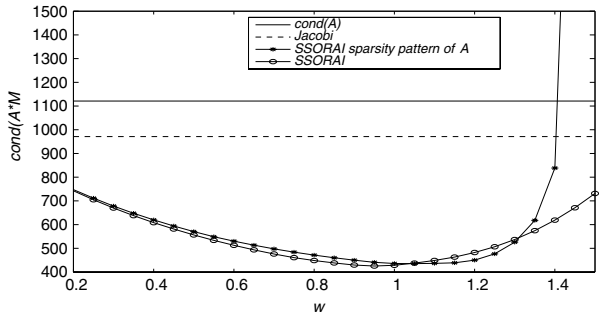

Fig. 1. Condition numbers versus ω for a positive definite matrix of size $n = 1105$.

The CSR format is one of the most popular storage format for sparse general matrices. It is particularly suitable for performing matrix–vector products.

4.2. CUDA

CUDA stands for *Compute Unified Device Architecture* and is a new hardware and software architecture for using NVIDIA *Graphics Processing Units* (GPUs) as a data-parallel computing device. CUDA is a parallel programming model [12,13] consists of a sequential *host program*, running on CPU host, and a *kernel program*, running on parallel GPU device. The host program sets up the data and transfers it to and from the GPU while the kernel program processes the data using a potentially large number of parallel threads. The threads of a kernel are grouped into a grid of *thread blocks*. The threads of a given block share a local store and may synchronize via barriers. Threads in different thread blocks cannot be synchronized. A modern NVIDIA GPU is built around an array of shared memory multiprocessors. Each multiprocessor is equipped with 8 scalar cores and 16 kB of high-bandwidth low-latency memory. The CUDA programming guide [13] provides tips for maximizing performance.

The parallelization of update operations (for x, r and p) is straightforward. In our code, update operations represent about 15% of GPU time. The crucial problem in the parallelization of the CG and PCG algorithms, on GPU, concerns the inner product and the matrix–vector multiplication.

The inner product seems inherently sequential but there is an efficient parallel algorithm even for GPU architecture: the *parallel prefix sum (scan)*. The GPU implementation of this algorithm is provided through Nvidia technical report [14].

Sparse matrix–vector operations represent the dominant cost in PCG algorithm (and in many iterative algorithm) for solving large-scale linear systems. If dense matrix–vector operations are regular and often limited by floating point throughput, sparse matrix–vector operations are much less regular in their access pattern and, consequently, are generally limited by bandwidth. In [15], Bell and Garland proposed two implementations of sparse matrix–vector multiplication for the CSR format: the first using one thread per row, and the second using 32-thread warp per matrix row. The latter gives best performances with a fine tuning of the number of warp threads.

Another implementation of the sparse matrix–vector multiplication $y = Ax$ is to split multiplication and addition operations as follows

$$z_{ij} = A_{ij}x_j, \quad \forall(i,j) \tag{4.1}$$

$$y_i = \sum_j z_{ij}, \quad \forall i. \tag{4.2}$$

5. Numerical experiments

We first investigate the behavior of our SSOR-AI preconditioner on a matrix arisen from a Poisson equation (2.1) (with $\alpha = 0$ and $\nu = 1$) on a unit disk discretized by finite element method. The resulting matrix A is symmetric, positive definite. The SSOR-AI preconditioner computed with the sparsity pattern of A is denoted by $\bar{M}_A$. We use MATLAB to approximate the condition number A, AD^{-1} (for Jacobi preconditioning), $A\bar{M}_A$ and $A\bar{M}$.

The first finite element mesh consists of $n = 1105$ nodes, the size of the unknown vector x. Fig. 1 shows the condition number against ω. We notice that the SSOR-AI preconditioner is significantly better than the Jacobi preconditioner [16].

We now compare our preconditioner with the IP preconditioner (3.6), proposed by [2]. The finite element mesh consists of $n = 8047$ nodes. Fig. 2 shows that with a suitable parameter ω, our approach ($\bar{M}$ and $\bar{M}_A$) has the best condition number. Fig. 2 also shows that $\omega = 1$ is not always the best choice for $\kappa(A\bar{M}_A)$ or $\kappa(A\bar{M})$. It can therefore be useful to adjust the parameter ω to reduce the condition number of $\kappa(A\bar{M}_A)$ or $\kappa(A\bar{M})$, rather than using $\omega = 1$.

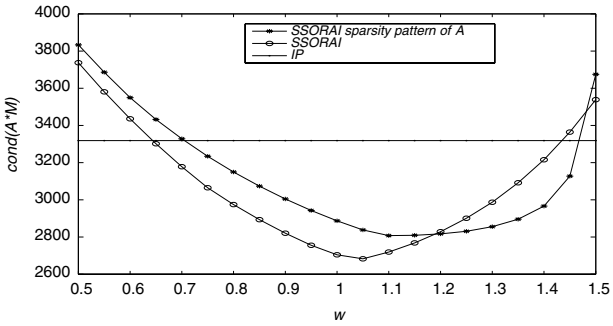

Fig. 2. Condition number of $A\bar{M}_A$, $A\bar{M}$ and $A\tilde{M}$ (IP preconditioner (3.6)) versus w for a positive definite matrix of size $n = 8047$; $\kappa(A) \approx 7607$ and $\kappa(AD^{-1}) \approx 6430$ (Jacobi).

Table 1
CG algorithm: CG-CPU vs. CG-GPU.

n	CG-CPU Time (s)	CG-GPU Time (s)	Ratio CG-CPU/CG-GPU
265 345	25.22	5.38	4.68
525 849	64.53	10.80	5.97
755 581	118.06	18.48	6.38
1 063 159	201.37	29.27	6.87
2 105 137	516.73	71.37	7.20

Table 2
Performances of the PCG algorithm, using 8-thread warp per row, and speed-up.

n	265 345	525 849	755 581	1 063 159	2 105 137
GPU Time (s)	4.09	8.46	13.59	20.63	50.37
Speed-up CG-GPU	1.29	1.27	1.35	1.41	1.41
Speed-up CG-CPU	6.16	7.62	8.68	9.76	10.25

Table 3
Performances of the PCG algorithm, using (4.1)–(4.2), and speed-up.

n	265 345	525 849	755 581	1 063 159	2 105 137
GPU Time (s)	2.86	7.79	13.90	22.57	62.05
Speed-up CG-GPU	1.88	1.38	1.32	1.29	1.15
Speed-up CG-CPU	8.81	8.28	8.49	8.92	8.32

In our numerical experiments, we use only $\bar{M}_A$ as preconditioner (with $\omega = 1.1$) to reduce the computational cost. Indeed, performing matrix–matrix multiplication is much less costly if the sparsity pattern of the resulting matrix is known. For the numerical experiments, we use the following processors:

CPU Intel Xeon Quad-Core 2.66 GHz, 12 GB RAM (using gFortran),
GPU NVIDIA Tesla T10, 240-core, 4 GB RAM (using CUDA).

The matrices are from discretization of the Poisson problem in unit disk with a finite element method. We use the sparse matrix–vector multiplication proposed in [15] but with 8-thread warp per matrix row, because the matrices have around 8 non-zero values per row.

In all algorithms, we start with $x_k = 0$ and we iterate until $\|r_k\| \, \|r_0\|^{-1} < 10^{-6}$.

In Table 1 we compare the CG algorithm implemented on GPU with its CPU counterpart. We can notice that without preconditioning, the CG algorithm on GPU is about 6 times faster than its CPU implementation.

In Table 2, we report the performances of the PCG algorithm using the sparse matrix–vector multiplication proposed in [15]. GPU Times include the computation of the SSOR-AI preconditioner (i.e. the sparse matrix–matrix multiplication). We note that the computational time required for computing $\bar{M}_A = \bar{K}^T\bar{K}$ is not significant, even for large matrices. For instance, for $n = 2\,105\,137$, the GPU time for performing $\bar{M}_A$ is 0.169 s. We notice that the preconditioning with SSOR-AI significantly improves the performance of the CG algorithm by about 30% in terms of computational time. The speed-up obtained with the PCG on GPU, with respect to the CPU implementation of the CG algorithm, then increases. For the largest problem, the PCG algorithm on GPU is 10 times faster than the CG algorithm on CPU.

In Table 3, we report the performances of the PCG algorithm using the sparse matrix–vector (4.1)–(4.2). We can notice that this version of the PCG algorithm is faster only for $n = 265\,345$ and $n = 525\,849$, the smallest problems.

6. Conclusion

We have presented a parallel implementation, on GPU, of the preconditioned conjugate gradient algorithm for linear systems with symmetric, positive definite matrix. Our preconditioner, derived from the standard SSOR, is an approximate inverse and can therefore be used in the PCG algorithm through a sparse matrix–vector multiplication. Numerical experiments show that the speed-up obtained with the PCG on GPU, with respect to the CPU implementation of the CG algorithm, is between 8 and 10 (depending on the sparse matrix–vector multiplication used).

We plan to investigate another SSOR approximate inverses (e.g. using second order approximation in (3.4)) as well as another sparse matrix storage format (e.g. *Block Compressed Sparse Row*). Our PCG algorithm is currently designed for 1-GPU platform. A multi-GPU implementation is under study to improve scalability.

Acknowledgments

This work was funded by *Bourse Innovation*, Conseil Régional d'Auvergne—UE Fonds FEDER.

References

[1] E.F. Kaasschieter, Preconditioned conjugate gradients for solving singular systems, J. Comput. Appl. Math. 24 (1988) 265–275.
[2] M. Ament, G. Knittel, D. Weiskopf, W. Strasser, A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform, in: Proc. 18th Euromicro Conference on Parallel, Distributed and NetWork-Based Computing, Pisa, Italy, February 17–19 2010, 583–592.
[3] G.H. Golub, C.F. Van Loan, Matrix Computations, The John Hopkins University Press, Baltimore, 1989.
[4] P. Lascaux, R. Théodor, Analyse Numérique Matricielle Appliquée à l'art de L'ingénieur, vol. 1, Masson, Paris, 1994.
[5] G. Meurant, Computer Solution of Large Systems, in: Studies in Mathematics and its Applications, North Holland, 1999.
[6] P. Lascaux, R. Théodor, Analyse Numérique Matricielle Appliquée à l'art de L'ingénieur, vol. 2, Masson, Paris, 1994.
[7] D. Luenberger, Linear and Nonlinear Programming, Addison Wesley, Reading, MA, 1989.
[8] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, 2003.
[9] E. Chow, Y. Saad, Approximate inverse preconditioners via sparse–sparse iterations, SIAM J. Sci. Comput. 19 (1998) 995–1023.
[10] M.J. Gorte, T. Huckle, Parallel preconditioning with sparse approximate inverses, SIAM J. Sci. Comput. 18 (1997) 838–853.
[11] I.S. Duff, A.M. Erisman, J.K. Reid, Direct Methods for Sparse Matrices, Clarendon Press, Oxford, 1986.
[12] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA, Queue 6 (2) (2008) 40–53.
[13] NVIDIA Corporation, NVIDIA CUDA programming guide, Technical Report, NVIDIA, 2008.
[14] M. Harris, Parallel prefix (scan) sum with CUDA, Technical Report, NVIDIA, 2008.
[15] N. Bell, M. Garland, Efficient sparse matrix–vector multiplication on CUDA, Technical Report NVR-2008-04, NVIDIA, 2008.
[16] L. Buatois, G. Caumon, B. Levy, Concurrent number cruncher: a GPU implementation of a general sparse linear solver, Int. J. Parallel Emergent Distrib. Syst. 24 (2009) 205–223.