NIH Public Access

Author Manuscript

Parallel Comput. Author manuscript; available in PMC 2011 June 1.

Published in final edited form as:

Parallel Comput. 2010 June 1; 36(5-6): 215–231. doi:10.1016/j.parco.2009.12.003.

GPU computing with Kaczmarz's and other iterative algorithms for linear systems

Joseph M. Elble[a], **Nikolaos V. Sahinidis**[b,1], and **Panagiotis Vouzis**[b]

Joseph M. Elble: elble@uiuc.edu; Nikolaos V. Sahinidis: sahinidis@cmu.edu; Panagiotis Vouzis: pvouzis@cmu.edu

[a] University of Illinois Urbana-Champaign, Department of Industrial and Enterprise Systems Engineering, Urbana, IL 61801

[b] Carnegie Mellon University, Department of Chemical Engineering, 5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract

The graphics processing unit (GPU) is used to solve large linear systems derived from partial differential equations. The differential equations studied are strongly convection-dominated, of various sizes, and common to many fields, including computational fluid dynamics, heat transfer, and structural mechanics. The paper presents comparisons between GPU and CPU implementations of several well-known iterative methods, including Kaczmarz's, Cimmino's, component averaging, conjugate gradient normal residual (CGNR), symmetric successive overrelaxation-preconditioned conjugate gradient, and conjugate-gradient-accelerated component-averaged row projections (CARP-CG). Computations are preformed with dense as well as general banded systems. The results demonstrate that our GPU implementation outperforms CPU implementations of these algorithms, as well as previously studied parallel implementations on Linux clusters and shared memory systems. While the CGNR method had begun to fall out of favor for solving such problems, for the problems studied in this paper, the CGNR method implemented on the GPU performed better than the other methods, including a cluster implementation of the CARP-CG method.

Keywords

Scientific computing on GPU; Linear systems; Iterative methods; Convex feasibility problem

1. Introduction

Natural and engineered systems are often modeled by sets of partial differential equations that have no closed-form solution. Discretization of these equations leads to large sparse systems of linear equations. A considerable amount of algorithmic and computational work has been performed to develop iterative algorithms for solving these systems as efficiently as possible [1].

[1]This work was supported in part by the Joint NSF/NIGMS Initiative to Support Research in the Area of Mathematical Biology under NIH award GM072023.

Publisher's Disclaimer: This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final citable form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

The current paper investigates the performance of several iterative solvers on a graphics processing unit (GPU). The algorithms we are interested in include Kaczmarz's sequential algorithm, as well as several block-parallel algorithms: Cimmino's method, component averaging (CAV), conjugate gradient normal residual (CGNR), symmetric successive overrelaxation (SSOR)-preconditioned conjugate gradient (CGMN), component-averaged row projections (CARP), and conjugate-gradient-accelerated CARP (CARP-CG).

Given the challenging nature of large-scale systems, parallel computing has been utilized in the development of efficient solution algorithms. In the context of GPUs, recent hardware developments, such as the NVIDIA Tesla C870, offer a massively multithreaded processor architecture that is ideal for high performance computing applications. These innovative designs have led many researchers to implement various linear algebra routines on the GPU. A Jacobi iterative solver was implemented on the GPU by Göddeke *et al.* [2]. Conjugate gradient and multigrid sparse matrix solvers were implemented on the GPU by Bolz *et al.* [3]. Krüger and Westermann [4] implemented direct solvers for sparse matrices, and studied their performance using multi-dimensional finite difference equations arising from the 2-D wave equation and the incompressible Navier-Stokes equations. An algorithm to solve dense linear systems using GPUs was implemented on an NVIDIA GeForce 7800 GPU by Galoppo *et al.* [5]. These are some of several studies recently reported using GPUs to accelerate the solution of various linear algebra problems.

There are many studies reporting the parallel solution of linear systems on hardware other than GPUs. Here, we mention those most closely related to the algorithms studied in this paper. Bramley and Sameh [6] implemented a block-sequential Kaczmarz algorithm with CG acceleration under five different partitioning schemes on structured grids. The disadvantage of the block-sequential approach is that it requires the identification of independent sets of equations which is difficult for unstructured grids. In [7], the same authors extended the work in [6] to include a CG acceleration of a block-Cimmino algorithm and a new projection method called V-RP. Their block-parallel CG-accelerated methods were shown to be robust in practice, but there was no clear "best" partitioning scheme. Arioli *et al.* [8] studied parallel CG acceleration of block-Cimmino for different block partitionings. The authors restricted their study to block-tridiagonal systems. Gordon and Gordon [9] introduced the CARP algorithm that divides the linear equations into blocks and operates in a block-parallel manner. Kaczmarz row projections are performed within each block in parallel and the results are then merged using component-averaging operations. CARP was shown to be very robust and suitable for unstructured grids.

The main purpose of the paper is to evaluate the performance on the GPU of several parallel algorithms with respect to solving large linear systems arising from the discretization of elliptic convection-diffusion partial differential equations. In particular, we are interested in identifying the best possible algorithm for this architecture, as well as in comparing the GPU against the CPU and earlier cluster implementations of these algorithms. For this purpose, we will investigate the performance of these algorithms on a set of problems addressed in many earlier works. This set of problems includes six partial differential equations proposed in [7], along with three additional partial differential equations that were investigated in [10,11]. All nine of these partial differential equations are strongly convection-dominated.

The remainder of the paper is organized as follows. Section 2 provides mathematical preliminaries and an introduction to GPU computing and architectures, including the NVIDIA Tesla C870, which is utilized for computations in this paper. The algorithms considered here are described in Section 3, while the philosophy and drivers behind our GPU implementation of these algorithms are detailed in Section 4. In Section 5, we present extensive computational results, followed by conclusions from this work in Section 6.

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

2. Preliminaries

2.1. Mathematical background and notation

The problem of finding a point in the intersection of two or more convex sets is referred to as the convex feasibility problem. Let $C_1, C_2, \ldots, C_n$ be closed convex subsets of a Hilbert space X with a nonempty intersection

$$C = C_1 \cap C_2 \cap \ldots \cap C_n \neq \emptyset.$$

The convex feasibility problem involves finding some x in C. In image reconstruction, the convex sets C_i are hyperplanes. The iterative algorithms presented in Section 3 use orthogonal projections on the hyperplanes C_i to solve these problems. A matrix $P \in \mathbb{R}^{n \times n}$ is an orthogonal projection onto $S \subseteq \mathbb{R}^n$ if $R(P) = S$, $P^2 - P = 0$, and $P^T = P$, where $R(P)$ is the range of P.

Define a linear system as $Ax = b$, where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and $b \in \mathbb{R}^m$. A consistent linear system is one where $b \in R(A)$. An inconsistent linear system is such that $b \notin R(A)$. Simply put, the equations of a linear system are consistent if they possess a common (but not necessarily unique) solution, and inconsistent otherwise.

A linear system with a low condition number is said to be *well-conditioned*, while a linear system with a relatively high condition number is said to be *ill-conditioned*. For a linear system with an invertible matrix, the condition number is given by

$$\kappa(A) = \left\| A^{-1} \right\| \left\| A \right\|,$$

where $\|A\|$ denotes the norm of A. Any norm, $\|\cdot\|$, without a subscript denotes the Euclidean norm. For rectangular matrices with full column rank, *i.e.*, $A \in \mathbb{R}^{m \times n}$ and $\text{rank}(A) = n$, the condition number is given by

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)},$$

where $\sigma_{\max}(A)$ denotes the maximum singular value of A and $\sigma_{\min}(A)$ denotes the minimum singular value of A.

Two nonzero vectors u and v are conjugate with respect to A if

$$u^T A v = \langle u, v \rangle_A = 0.$$

Likewise, it is necessary to define the notation

$$\left\| x \right\|_S^2 = x^T S x.$$

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

When discussing an iterative algorithm, the *k*th iterate of any vector will always be denoted by a superscript in parenthesis, *e.g.*, $x^{(k)}$. On the other hand, the position in the vector will be denoted with a subscript. For example, $x_j^{(k)}$ would denote the *j*th element of the *k*th iteration of x. For an $m \times n$ matrix A, a_{ij} denotes the element of A in the *i*th row and *j*th column. Furthermore, a_i is used to denote the *i*th row vector of A, unless otherwise noted. The standard basis vectors are written e_i.

2.2. The graphics processing unit

This subsection briefly describes the architecture of a GPU from the perspective of a programmer using the *compute unified device architecture* (CUDA) [12]. CUDA is a hardware and software architecture that issues and manages data-parallel computations on a GPU. The GPU is a single-instruction multiple-data (SIMD) parallel device. The GPU should be viewed as a *compute device* or coprocessor, while the CPU should be viewed as the host. Since the GPU is a SIMD architecture with a host device, the GPU is utilized for data-parallel and computationally intensive portions of the algorithm or application.

The computationally intensive portions of the algorithm are executed in parallel using thousands of *threads*. Each thread executes the same set of instructions independently on different data. The GPU instructions are referred to as a *kernel*, which is downloaded to the device. A *block* is a group of threads that share data through *shared memory* and synchronize their execution to coordinate their memory accesses. These synchronization points are specified within the kernel and act as a barrier where all threads in the block are suspended until they reach this barrier. A *grid* is an organization of thread blocks. Within each grid, there exist a programmer-defined number of blocks.

The memory model for the GPU is provided at the grid level. Read-write access to global memory and read-only access to constant and texture memory is afforded to the entire grid. The host has read-write access with global, constant, and texture memory. Each thread has read-write access to its own set of registers and local memory, while each block has read-write access to designated shared memory.

The multiprocessors on an NVIDIA Tesla C870 are organized in a *streaming processor array*. Each element of this array is referred to as a *texture processor cluster*. In the case of the NVIDIA Tesla C870, there are eight texture processor clusters. A texture processor cluster consists of *texture memory* and two *streaming multiprocessors*. Each streaming multiprocessor contains instruction and data cache, instruction fetch and dispatch unit, shared memory, eight *streaming processors*, and two *special function units*. The special function units allow for fast single-precision mathematical computations, such as sine, cosine, logarithm, and exponential.

The interested reader can find additional discussions on the Tesla architecture in [13]. More detailed information on GPU computing can be found in two recent special issues of *IEEE Proceedings* [14] and the *Journal of Parallel and Distributed Computing* [15].

3. Iterative methods

This section describes the iterative methods utilized in this paper. Interest is placed on solving linear systems of the form $Ax = b$, where A is an $m \times n$ matrix, x is an n-vector, and b is an m-vector.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

3.1. Kaczmarz's algorithm

Kaczmarz introduced this algorithm in [16]. Kaczmarz's approach is a projection method that is also referred to as ART (algebraic reconstruction technique) and is used for solving linear systems from image reconstruction problems.

Kaczmarz's algorithm is sequential. The algorithm sweeps through rows of A in a cyclic manner. At each iteration, the previous iterate is projected orthogonally onto the solution hyperplane $\langle a_i, x\rangle = b_i$. This orthogonal projection leads to the normalized step at iteration k

$$s^{(k)}=\lambda_i \frac{b_i - \langle a_i, x^{(k)}\rangle}{\|a_i\|^2} a_i, \tag{1}$$

where λ_i is a cyclic relaxation parameter that extends the projections either in front of the hyperplane ($\lambda_i < 1$), exactly on the hyperplane ($\lambda_i = 1$), or beyond the hyperplane ($\lambda_i > 1$), and we assume henceforth that $0 < \lambda < 2$. In this case, i is equal to k modulus $m + 1$, with $k \geq 0$. A randomized version of the algorithm has also been proposed in [17], where the row i is chosen at random rather than sequentially. In either case, each set of m iterations is referred to as a *sweep*. The iterate progresses as follows

$$x^{(k+1)}=x^{(k)}+s^{(k)}$$

where the step $s^{(k)}$ is defined by (1).

Algorithm 1 is a presentation of Kaczmarz's method.

3.2. Cimmino's algorithm

Cimmino introduced his algorithm in [18]. This method is highly parallel and guaranteed to converge in the inconsistent case. In practice, the method is slow, requiring many iterations to reduce the residual error and approach the solution. The algorithm converges to the weighted least-squares solution,

which minimizes the weighted sum of the squares of the distances to the sets $C_1, \ldots, C_n$ discussed in Subsection 2.1. This result is due to Combettes [19].

The method of Cimmino involves a simultaneous orthogonal projection onto the set of solution hyperplanes $\langle a_i, x\rangle = b_i$, $i = 1, \ldots, m$. The orthogonal projections lead to a normalized step at iteration k

$$s_j^{(k)}=\frac{\lambda_k}{m}\sum_{i=1}^{m}\frac{b_i - \langle a_i, x^{(k)}\rangle}{\|a_i\|^2} a_{ij}, \quad j=1,\ldots,n, \tag{2}$$

where $s_j^{(k)}$ is the jth element of the kth normalized Cimmino step. The observant reader should recognize that each $s_j^{(k)}$ can be computed in parallel for all $j = 1, \ldots, n$. The iterate progresses as before:

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

$$x^{(k+1)}=x^{(k)}+s^{(k)}.$$

This algorithm can also be expressed in matrix form as:

$$x^{(k+1)}=x^{(k)}+\lambda_k A^T D\left(b-Ax^{(k)}\right),$$

where

$$D=\frac{1}{m}\mathrm{diag}\left(\frac{1}{\|a_1\|^2},\frac{1}{\|a_2\|^2},\ldots,\frac{1}{\|a_m\|^2}\right). \tag{3}$$

Algorithm 2 is a presentation of Cimmino's algorithm.

3.3. Component-averaging

Component-averaging (CAV) was introduced by Censor *et al.* [20]. CAV projects the current iterate onto all the system's hyperplanes in parallel, just like Cimmino's algorithm. CAV retains the desired convergence properties of Cimmino's method, in the sense that it converges in the inconsistent case. Furthermore, CAV exhibits significantly faster numerical convergence for large sparse systems in practice. The exhibited acceleration is noticeable on a single processor as well as parallel architectures.

The scalar weighting of Cimmino's method is replaced by diagonal sparsity weighting. That is, the weights are inversely proportional to the number of nonzeros in each column. CAV's orthogonal projections lead to a normalized step at iteration k

$$s_j^{(k)}=\frac{\lambda_k}{m}\sum_{i=1}^{m}\frac{b_i-\left\langle a_i,x^{(k)}\right\rangle}{\sum_{p=1}^{n}s_p a_{ip}^2}a_{ij},\quad j=1,\ldots,n, \tag{4}$$

where s_p is the number of nonzeros in column p, for $p = 1, \ldots, n$. As with the previously introduced iterative methods, the iterate progresses using the formula

$$x^{(k+1)}=x^{(k)}+s^{(k)}.$$

As in Cimino's algorithm, CAV can be expressed in matrix form

$$x^{(x+1)}=x^{(k)}+\lambda_k A^T D_s\left(b-Ax^{(k)}\right),$$

where

$$D_s = \mathrm{diag}\left(\frac{1}{\|a_1\|_s^2}, \frac{1}{\|a_2\|_s^2}, \ldots, \frac{1}{\|a_m\|_s^2}\right). \quad (5)$$

The reader will notice only a subtle difference between CAV and Cimmino's method, namely (3) versus (5), that is, the difference between the sparsity-oriented versus scalar weighting schemes which are captured by equations (3) and (5).

Lastly, CAV is proven to converge to a minimizer of a certain proximity function regardless of x_0 and independent of the consistency of the underlying linear system. For more details on these convergence results, the reader is referred to [20].

Algorithm 3 is a presentation of CAV.

3.4. Conjugate gradient normal residual (CGNR)

The CGNR algorithm was introduced by Hestens and Stiefel in [21]. In order to solve the linear system $Ax = b$ when A is nonsymmetric, CGNR attempts to solve the equivalent system

$$A^T Ax = A^T b. \quad (6)$$

By construction, this system is symmetric positive semidefinite. CGNR works by applying the conjugate gradient method to the normal equations (6). Conjugate gradient can be applied to (6) because $A^T A$ is symmetric positive semidefinite.

The algorithm does not store $A^T A$ explicitly in memory. This fact makes CGNR a natural choice for sparse matrices, since operations such as matrix-vector multiplication are usually very efficient. In addition, the implicit use of $A^T A$ leads to relatively large diagonal elements. This contributes to the algorithm's relative robustness.

The downside of using CGNR is the fact that $\kappa(A^T A) = \kappa(A)^2$. The relatively large condition number of (6) leads to a slow rate of convergence, depending, of course, on the condition number of the original system. This important fact makes a preconditioner all the more important for this algorithm.

Algorithm 4 is a presentation of the CGNR algorithm.

3.5. Symmetric successive overrelaxation (SSOR)-preconditioned conjugate gradient (CGMN)

CGMN is a conjugate gradient acceleration of the SSOR algorithm. CGMN was introduced by Björck and Elfving in [22], where it was referred to as an SSOR preconditioning of CG. It will become clear that CGMN can also be considered a CG acceleration of Kaczmarz's algorithm. Similar to CGNR, it is advantageous to utilize CGMN when A is sparse. CGMN is a CG acceleration of the SSOR algorithm applied to the system

$$x = A^T y, AA^T y = b. \quad (7)$$

Therefore, a relatively large condition number can negatively impact the rate of convergence since $\kappa(A^T A) = \kappa(A)^2$.

NIH-PA Author Manuscript NIH-PA Author Manuscript NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Define a Kaczmarz sweep as before

$$x^{(i+1)}=x^{(i)}+\lambda_i \frac{b_i - a_i^T x^{(i)}}{\|a_i\|^2} a_i, \quad i=1,\ldots,m. \tag{8}$$

Applying Gauss-Seidel's method to (7), it is evident that the ith minor step, $y^{(i)}$, is updated by

$$s_y^{(i)}=e_i\left(b_i - a_i^T A^T y^{(i)}\right)/a_i^T a_i.$$

By $x = A^T y$,

$$\begin{aligned} x^{(i+1)}&=x^{(i)}+A^T s_y^{(i)} \\ &=x^{(i)}+A^T e_i \frac{(b_i-a_i^T A^T y^{(i)})}{a_i^T a_i} \\ &=x^{(i)}+\frac{a_i(b_i-a_i^T x^{(i)})}{\|a_i\|^2}. \end{aligned}$$

Therefore, Gauss-Seidel's method is equivalent to Kaczmarz's method for (7).

Since CGMN involves an SSOR relaxation parameter, (8) becomes

$$x^{(i+1)}=x^{(i)}+\lambda \frac{b_i - a_i^T x^{(i)}}{\|a_i\|^2} a_i, \quad i=1,\ldots,m, \tag{9}$$

where λ is the relaxation parameter. If the relaxation parameter is $0 < \lambda < 2$, then the algorithm is consistent with Kaczmarz's method. SSOR implies a forward and backward sweep of Kaczmarz. Therefore, (9) becomes

$$x^{(i+1)}=x^{(i)}+\lambda \frac{b_j - a_j^T x^{(i)}}{\|a_j\|^2} a_j, \quad i=1,\ldots,2m; j=\min(i, 2m+1-i).$$

It is now evident that the forward sweep is followed by the backward sweep. Furthermore, the double sweep can be transformed into

$$x^{(k+1)}=Q_{\text{SSOR}} x^{(k)}+R_{\text{SSOR}} b,$$

where

$$Q_{\text{SSOR}}=Q_1 Q_2 \ldots Q_m Q_m \ldots Q_2 Q_1,$$

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

and

$$Q_i = I - \lambda a_i a_i^T / \|a_i\|^2.$$

Q_{SSOR} is symmetric, since Q_i is symmetric for all i.

Conjugate gradient is then applied to the system

$$(I - Q_{\mathrm{SSOR}})x = R_{\mathrm{SSOR}} b.$$

Define

$$c = R_{\mathrm{SSOR}} b \text{ and } B = (I - Q_{\mathrm{SSOR}}).$$

Then, the conjugate gradient method is applied to solve the symmetric positive semidefinite system $Bx = c$, where B is symmetric and positive semidefinite and c is in the range of B. For an arbitrary $x^{(0)}$, define

$$\begin{gathered}
p^{(0)} = r^{(0)} = c - Bx^{(0)}, \\
q^{(k)} = Bp^{(k)}, \\
\alpha^{(k)} = \|r^{(k)}\|^2 / (p^{(k)})^T q^{(k)}, \\
x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}, \\
r^{(k+1)} = r^{(k)} - \alpha^{(k)} q^{(k)}, \\
\beta^{(k)} = \|r^{(k+1)}\|^2 / \|r^{(k)}\|^2 \\
p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}.
\end{gathered}$$

The algorithm proceeds by computing $x^{(k+1)}$, $r^{(k+1)}$, and $p^{(k+1)}$ for each $k = 0, 1, 2, \ldots$, where $q^{(k)} = p^{(k)} - Q_{\mathrm{SSOR}} p(k)$. To compute $p^{(0)}$, it is necessary to perform a full forward and backward Kaczmarz sweep as defined by (8) on the system $Bx = c$. To compute $q^{(k)}$, let $x^{(1)} = p^{(k)}$ and again use the forward and backward Kaczmarz sweep with $b = 0$, which leads us to $q^{(k)} = p^{(k)} - Q_{\mathrm{SSOR}} p(k)$. Notice that there is no R_{SSOR} term since $b = 0$.

Algorithm 5 is a presentation of CGMN.

3.6. Conjugate-gradient-accelerated component-averaged row projections

Component-averaged row projections (CARP), which was introduced by Gordon and Gordon [10], divides the set of m equations into blocks. Each processor performs Kaczmarz iterations in parallel. The results of these Kaczmarz iterations are then merged together by averaging to form the next iterate.

In [10], the authors proved that CARP is equivalent to Kaczmarz's algorithm with cyclic relaxation parameters in some superspace $\mathbb{R}^s$, where $s = \sum_{j=1}^{n} s_j$ and s_j denotes the number of nonzeros in column j. There is a mapping that exists between $\mathbb{R}^s$ and $\mathbb{R}^n$, such that every $x \in \mathbb{R}^n$ maps to some $y \in \mathbb{R}^s$. That is, every component x_j of x maps to s_j components of y.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

The fact that CARP is equivalent to Kaczmarz's algorithm in some superspace $\mathbb{R}^s$ allows for the construction of a CG-accelerated CARP. The CG acceleration is performed in much the same manner as the CG acceleration of SSOR, which created CGMN. A forward followed by a backward sweep of CARP is performed, rather than the forward and backward sweep of Kaczmarz. Due to the superspace equivalency, it is shown in [10] that the CG acceleration of CARP always converges, even when the linear system is inconsistent and/or rectangular. The details are available in [10].

Algorithm 6 is a presentation of CARP-CG.

4. GPU implementation

4.1. Preliminary tests

The parallelization of linear algebra algorithms has been under investigation for the last few decades, and led to the proposal of a number of implementations that utilize parallel computers. Depending on the specific parallel architecture, there are appropriate algorithms that match the characteristics of the underlying hardware such as memory organization and size, node communication bandwidth, and processing model (MIMD, SIMD). In many cases, a specific algorithm does not lend itself to a particular architecture, and it is required to develop a parallel version of the algorithm to achieve the desired speedup.

With this in mind, Gordon and Gordon [9] proposed the CARP algorithm, a block-parallel version of the Kaczmarz algorithm implemented on a 16-node Linux cluster. In addition to exhibiting increased robustness for the problems studied in [9], CARP also exhibited a better or comparable solution time relative to other well-established algorithms that had been proposed for the same kind of problems. In implementations with a single general-purpose processor, CARP and CARP-CG, being block-parallel algorithms, are not suitable; thus Gordon and Gordon in [11] show that the CGMN algorithm gives the best robustness and performance, with CGNR offering the same robustness but lower performance on the same set of problems as in [10]. Note that CARP-CG reduces to CGMN, when the number of blocks is equal to one.

In the preliminary search of the most appropriate iterative algorithms to implement on the GPU, it became clear to us that Kaczmarz's algorithm may not be the best possible algorithm for the GPU architecture. We first investigated Kaczmarz's algorithm in the form presented in [16]. It is important to keep in mind that while a thread is very cheap to originate on a GPU, allowing for performance gains in operations as simple as level-one BLAS routines, the problem must be large enough to mitigate the overhead of launching the thread, transferring the data to and from the GPU, and any interruptions necessary to check for convergence on the CPU. It was quickly discovered that, due to this fact and because the only available operations of this algorithm that can be parallelized are the dot product and vector addition in (1), this algorithm could only exhibit superior performance on the GPU when studying very large, dense linear systems. For details on how one can use a GPU to implement reduction-based parallel operations and other BLAS routines, the reader is referred to [12]. The performance of the implementation is shown in Table 1. In this table, the problems studied are one hundred percent dense $n \times n$ systems, where the elements of A and b are drawn randomly from a normal distribution with a mean of zero and standard deviation of one. The reader can clearly see that the memory initialization and memory copy necessary to load the data on the GPU can be costly for dense systems depending on the total number of iterations required. However, this cost is amortized by the superior cost per iteration as the problem size grows and/or the number of required iterations increases.

NIH-PA Author Manuscript NIH-PA Author Manuscript NIH-PA Author Manuscript

In the case of Cimmino's algorithm, we quickly discovered two facts. First, this algorithm exhibited very poor solution times due to the number of necessary iterations for the particular PDEs studied in this paper. Additionally, the algorithm was very costly at each iteration, regardless of whether it was implemented on the GPU or the CPU. We implemented Cimmino's algorithm for dense linear systems in order to get a better idea of the best-case speedup that could be expected by placing this algorithm on the GPU. The results are presented in Table 2. Cimmino can be parallelized on the GPU by implementing slightly specialized level-two BLAS routines. For more details on how one might implement level-two BLAS routines for dense matrices, the reader is referred to [12]. Ultimately, the low potential for improvement in the cost per iteration and the number of required iterations led us to reject this method for the particular PDE problems studied.

In addition, CAV was studied. Due to the fact that CAV is equivalent to Cimmino's algorithm when applied to dense problems, CAV was preliminarily tested on the GPU using banded linear systems (also generated randomly). We found that the method had potential and mapped well to the GPU. However, as was the case with Cimmino's algorithm, it appeared that CAV requires too many iterations. Based on the results demonstrated in [20], albeit on different problems, and the potential for speedup on the GPU (see Table 3 and Table 4), we decided it was appropriate to at least try this method on the PDEs in Section 5. In Table 3, the number of diagonals above and below the main diagonal were held constant at twenty-five, while the size of the matrix itself was varied. In Table 4, the size of the matrix was fixed at $25,000 \times 25,000$, while the number of nonzero diagonals above and below the main diagonal varied from one to sixty-four. The relatively poor convergence witnessed here is due to the small variance in the nonzero elements of the PDEs studied. This relatively small variance causes CAV to perform no better than Cimmino.

4.2. GPU implementation

We have tailored the kernel to the seven bands at fixed offsets of the A matrix. The reads and writes can be perfectly coalesced with the restriction that the number of the discretization points on the x direction is a multiple of 16 to achieve coalesced global memory accessing. In other words, if we desire 40 discretization points on the x axis, it is better to choose 48, which will give higher resolution and better performance than the 40-point discretization.

For a desired tolerance of 10^{-7}, the arithmetic results need to be at least double precision. There are two ways to achieve this accuracy. The first is to exclusively use double-precision floating-point arithmetic for all operations. The second is to use single-precision floating point with iterative refinement in order to achieve the required double-precision termination criterion.

Using exclusively double precision for all operations can be necessary depending on the size of the problem, the algorithm that is used, and the termination criterion. The drawback of using double-precision floating point is the lower speed of the floating-point unit relative to single-precision, and the requirement to load and store 64-bit data, compared to the 32-bit data of single-precision floating point. When it is necessary or desired to use single-precision arithmetic, iterative refinement reduces the roundoff errors in the computed solution of a system of linear equations. The basic steps of solving $Ax = b$ with iterative refinement are given in Algorithm 7.

Since the Tesla C870 GPU is only capable of single precision, the double-precision refinement is done on the CPU, which involves the transfer of the data from the GPU to the CPU and *vice versa* in each refinement iteration. This extra data transfer is unavoidable with the Tesla C870. Newer GPU models, such as the Tesla C1060 have double- and single-precision capabilities, which would eliminate these data transfers. In our current implementation, the single-precision step of the algorithm is carried out on the GPU and involves the solution of $Ac = d$ with one

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

of the iterative methods CGNR, CGMN, CARP-CG, or CAV. At the end of each internal iteration, we test whether the solution of $Ac = d$ has been improved or not by checking whether $||c^{(k)}||/||c^{(k-1)}|| > 1.0$. In case this condition is not met, the $Ac = d$ loop is terminated and the iterative-refinement step updates the solution of $x = x + c$ in double precision.

Even though a double-precision GPU could carry out all the operations in double precision, thereby avoiding the refinement steps, it is not guaranteed that the overall performance would be better than using iterative refinement [23]. Although the exclusive use of double precision requires fewer iterations to achieve convergence, this performance benefit is assuaged by the increased memory bandwidth and the reduced performance of double-precision arithmetic. For example, it is notable that the peak single-precision performance of the Tesla C1060 is 933 GFLOPS, while its peak double-precision performance is only 78 GFLOPS.

5. Computational results

The test problems used here are the same used in [9,10,11], and are given below:

1. $\Delta u + 1000 = F$
2. $\Delta u + 1000e^{xyz}(u_x + u_y - u_z) = F$
3. $\Delta u + 100xu_x - yu_y + zu_z + 100(x + y + z)u/xyz = F$
4. $\Delta u - 10^5x^2(u_x + u_y + u_z) = F$
5. $\Delta u - 1000(1 + x^2)u_x + 100(u_y + u_z) = F$
6. $\Delta u - 1000[(1 - 2x)u_x + (1 - 2y)u_y + (1 - 2z)u_z] = F$
7. $\Delta u - 1000x^2u_x + 1000u = F$
8. $\Delta u - \partial(10e^{xy}u)/\partial x - \partial(10e^{-xy}u)/\partial y = F$
9. $\Delta u - \partial(1000e^{xy}u)/\partial x - \partial(1000^{e-xy}u)\partial y = F$

Here, we use the notation $\Delta = u_{xx} + u_{yy} + u_{zz}$. Problems 1–7 have the following analytical solution:

$$\begin{array}{c}\text{Problem 1:}\quad u(x,y,z)=xyz(1-x)(1-y)(1-z)\\ \text{Problem 2:}\quad u(x,y,z)=x+y+z\\ \text{Problem 3 - 7:}\quad u(x,y,z)=e^{xyz}\sin(\pi x)\sin(\pi y)\sin(\pi z).\end{array}$$

The expression for the right-hand side function F in problems 1–7 is computed analytically, using the preassigned solution. For problems 8 and 9, the right-hand-side function F is irrelevant, because the equation system was set up by first computing the system matrix A, and then computing $b = Av$, where v was chosen as $v = (1, 1,\ldots, 1)^T$ (similar to the approach taken in [1, Problem F3D]). All the test problems were solved on the unit cube domain $[0, 1] \times [0, 1] \times [0, 1]$, with the Dirichlet boundary condition $u = 0$. The problems were discretized using a uniform mesh with the same number of mesh points in each direction, and the equations were obtained by using a seven-point centered difference scheme. Test runs were made for problems of size $80 \times 80 \times 80 = 512,000$ equations. Additional tests were also made on smaller grid sizes in order to study how the various algorithms perform as the mesh is gradually refined.

Since the results in this section are compared to the results in [10,11], an identical termination criterion is chosen, which is based on the relative residual: $\text{res}/\text{res}^{(0)} = ||b - Ax||/||b - Ax^{(0)}|| < 10^{-7}$, with $x^{(0)} = 0$. For problems 3 and 7, this criterion was unattainable. Instead, we used the same termination criteria used in [10,11], *i.e.*, 10^{-4} and $5 \cdot 10^{-4}$, respectively. Since this

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

criterion depends on the scaling of the equations, we first normalized the equations (for all the tested methods) by dividing each equation by the L_2-norm of its coefficients.

We investigate the GPU implementation of the most robust methods (CGNR, CGMN, CARP-CG) for the nine PDE problems, and we compare their performance to a single CPU implementation of CGNR and CGMN, and to the 16-node cluster implementation of CARP-CG [10]. The CARP-CG is a block-parallel algorithm and is suitable only for parallel computers, thus its CPU implementation is not considered. Moreover, we show the results of the GPU implementation of the CAV algorithm for the sparse PDE problems due to the fact that CAV achieves the highest GFLOPS performance.

Figures 1–9 present the convergence of each method for each problem with respect to time. It is clear from these figures that, for all problems, the GPU implementation of CGNR achieves the best performance relative to the other methods on both the CPU and the GPU. On the CPU, CGMN is consistently better than CGNR, as expected [11]. Although CARP-CG performs the best on a 16-node Linux cluster [10], the proposed implementation does not map well on the GPU architecture. The same can be observed for CGMN, which lends itself to a single CPU architecture, but is not the best choice for a GPU implementation. The relaxation parameters, λ, for CGMN are identical to the ones in [11].

In general, GPUs are suitable for algorithms, such as BLAS, that exhibit fine grain parallelism. There is a trade-off between using small blocks to take advantage of the fine grain parallelism available on the GPU and the number of iterations required, since one of the basic underlying assumptions in CARP-CG is that the block boundary points are just a small fraction of the total number of points [10]. The proposed GPU CARP-CG is a direct implementation of the algorithm of [10] with uniformly divided grids and scaling-factor values that give the smallest number of iterations. The Kaczmarz sweeps can be executed on the GPU or on the CPU. Despite the additional burden of transferring the data to the CPU, executing the Kaczmarz sweeps on the CPU is a better choice. For the same reason, CGMN does not map well on the GPU, as the Kaczmarz sweeps slow down the performance of the algorithm. CAV is the algorithm that achieves the highest GFLOPS performance on the GPU, as can be seen in Table 5. However, its extremely slow rate of convergence makes CAV the slowest algorithm among the ones studied in this paper.

The algorithm that exhibits the best balance between parallelization and rate of convergence with iterative refinement is CGNR. In Tables 6 and 7, we display the number of iterations and times to solution, respectively, for each architecture (GPU, CPU, 16-node cluster) and algorithm. Although the GPU CGNR requires more iterations to achieve the same level of precision relative to GPU CGMN and GPU CARP-CG, we observe that GPU CGNR exhibits the best solution time due to its superior utilization of the fine-grain parallelism available on the GPU architecture. The very large number of iterations required by GPU CAV results in its poor performance and its failure to converge for Problems 2, 4, and 8.

The robustness of the algorithms of [10,11] is preserved here despite the single-precision arithmetic, largely due to iterative refinement, which guarantees convergence to a solution within the desired accuracy. Moreover, the CGNR algorithm with single-precision iterative refinement is much more robust than CGMN and CARP-CG, in terms of required iterations to convergence. In Table 8, we present the ratio of iterative-refinement iterations for each algorithm over the double-precision iterations. It can be observed from this table that CGNR is affected less by the use of single-precision floating-point arithmetic. This is demonstrated further in Table 9, where it can be seen that the number of iterative-refinement iterations is always smaller for CGNR in every problem except for Problem 5. Moreover, CGNR does not involve a relaxation parameter that needs to be adjusted as in CARP-CG and CGMN.

NIH-PA Author Manuscript NIH-PA Author Manuscript NIH-PA Author Manuscript

Finally, in Figure 10, we show the speedup obtained by the GPU in comparison to the CPU CGMN and the 16-node cluster CARP-CG implementations. The GPU implementation is 5–20 times faster than the CPU, depending on the particular problem. Compared to the 16-node cluster, the GPU implementation is 1–3 times faster. The achieved speedup for CGNR is roughly a factor of 30 to 40.

6. Conclusions

Among the algorithms considered in this paper, CGNR was shown to be the most efficient for solving the investigated partial differential equations on the GPU. This finding is in contrast to earlier works on the CPU, where CGMN was found to be consistently better than CGNR. Moreover, CGNR was the most robust method under the GPU's single-precision floating-point arithmetic. CAV was able to achieve the highest GFLOPS but slowest solution time due to poor convergence.

The GPU has a clear advantage when compared to the CPU. In particular, the computational results demonstrated that the GPU implementation is five to twenty times faster than the CPU, depending on the particular problem. Compared to a 16-node cluster, the GPU implementation was one to three times faster. When the cost of the hardware, its power consumption, and its portability are accounted for, it becomes apparent that a single GPU offers a clear advantage over a 16-node Linux cluster.

While algorithm performance is known to be architecture-specific, our computational results specifically demonstrate that the GPU offers a low-cost and high-performance computing solution for solving large-scale partial differential equations. Continued improvements, such as performance and double precision, will only make the GPU more attractive as a high-performance computing device for scientific computing applications.

Acknowledgments

We would like to thank Dan Gordon for detailed suggestions that helped us considerably improve the quality of this paper.

References

1. Saad, Y. Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics; Philadelphia, PA: 2000.
2. Göddeke D, Strzodka R, Turek S. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, International Journal of Parallel. Emergent and Distributed Systems 2007;22(4):221–256.
3. Bolz J, Farmer I, Grinspun E, Schröder P. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. ACM Transactions on Graphics 2003;22:917–924.
4. Krüger J, Westermann R. Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics 2003;22:908–916.
5. Galoppo, N.; Govindaraju, NK.; Henson, M.; Manocha, D. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. Proceedings of the Conference on Supercomputing; 2005. p. 3
6. Bramley R, Sameh A. Domain decomposition for parallel row projection algorithms. Applied Numerical Mathematics 1991;8(4–5):303–315.
7. Bramley R, Sameh A. Row projection methods for large nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing 1992;13(1):168–193.
8. Arioli M, Duff I, Noailles J, Ruiz D. A block projection method for sparse matrices. SIAM Journal on Scientific and Statistical Computing 1992;13(1):47–70.

NIH-PA Author Manuscript NIH-PA Author Manuscript NIH-PA Author Manuscript

9. Gordon D, Gordon R. Component-averaged row projections: A robust, block-parallel scheme for sparse linear systems. SIAM Journal on Scientific Computing 2005;27(3):1092–1117.
10. Gordon, D.; Gordon, R. Tech rep. Department of Computer Science, University of Haifa; Haifa, Israel: Apr. 2008 CARP-CG: A robust and efficient parallel solver for linear systems, applied to strongly convection-dominated elliptic partial differential equations. submitted for publication
11. Gordon D, Gordon R. CGMN revisited: Robust and efficient solution of stiff linear systems derived from elliptic partial differential equations. ACM Transactions on Mathematical Software 35(3)
12. http://www.nvidia.com/object/cudahome.html
13. Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro 2008;28:39–55.
14. Lin, C.; Manocha, D., editors. IEEE Proceedings. Vol. 96. 2008. Special Issue on Cutting-Edge Computing: Using New Commodity Architecture; p. 753-899.
15. Gottlieb, A.; Hwang, K.; Sahni, S., editors. Journal of Parallel and Distributed Computing. Vol. 68. 2008. Special Issue on General-Purpose Processing Using Graphics Processing Units; p. 1305-1402.
16. Kaczmarz S. Angenäherte auflösung von systemen linearer gleichungen. Bulletin de l'Academie Polonaise des Sciences Letters 1937;35:355–357.
17. Strohmer, T.; Vershynin, R. Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques. Springer; 2006. A randomized solver for linear systems with exponential convergence; p. 499-507.
18. Cimmino G. Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari. La Ricerca Scientifica XVI 1938;9:326–333.
19. Combettes PL. Inconsistent signal feasibility problems: Least-squares solutions in a product space. IEEE Transactions on Signal Processing 1994;SP-42:2955–2966.
20. Censor Y, Gordon D, Gordon R. Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems. Parallel Computing 2001;27(6):777–808.
21. Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 1952;49:409–435.
22. Björck Å, Elfving T. Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. BIT Numerical Mathematics 1979;19:145–163.
23. Göddeke, D.; Strzodka, R. Tech rep. Technical University Dortmund; 2008. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double precision GPUs).

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

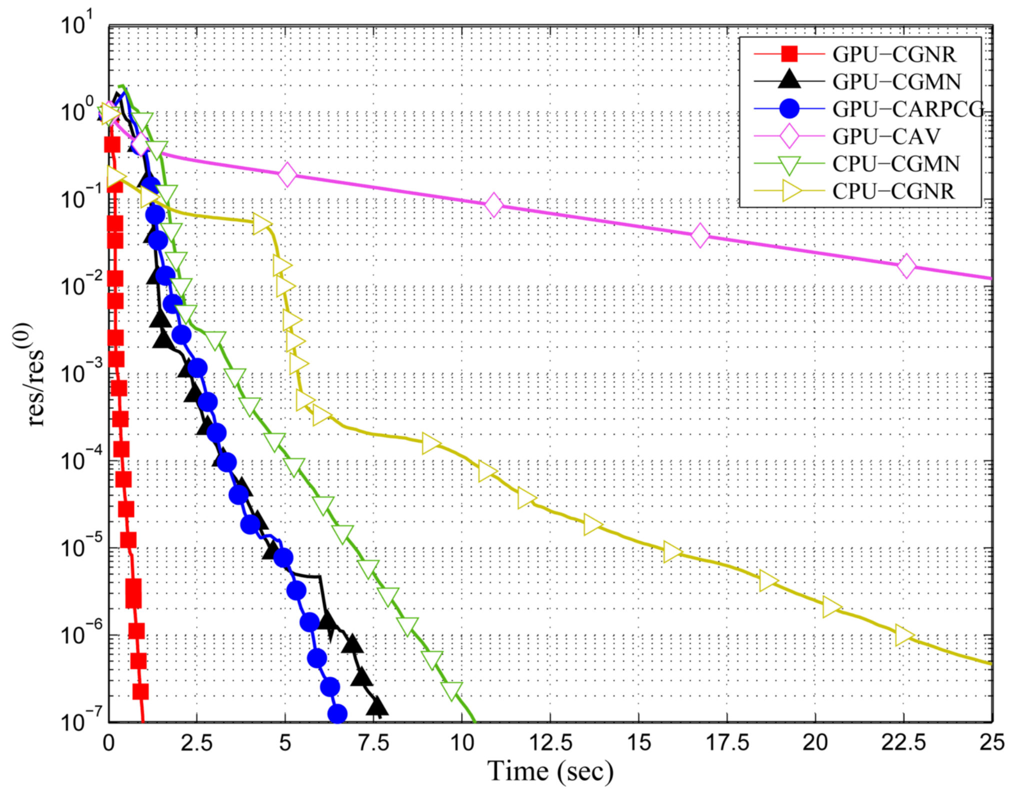

Figure 1.

Convergence results for Problem 1.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

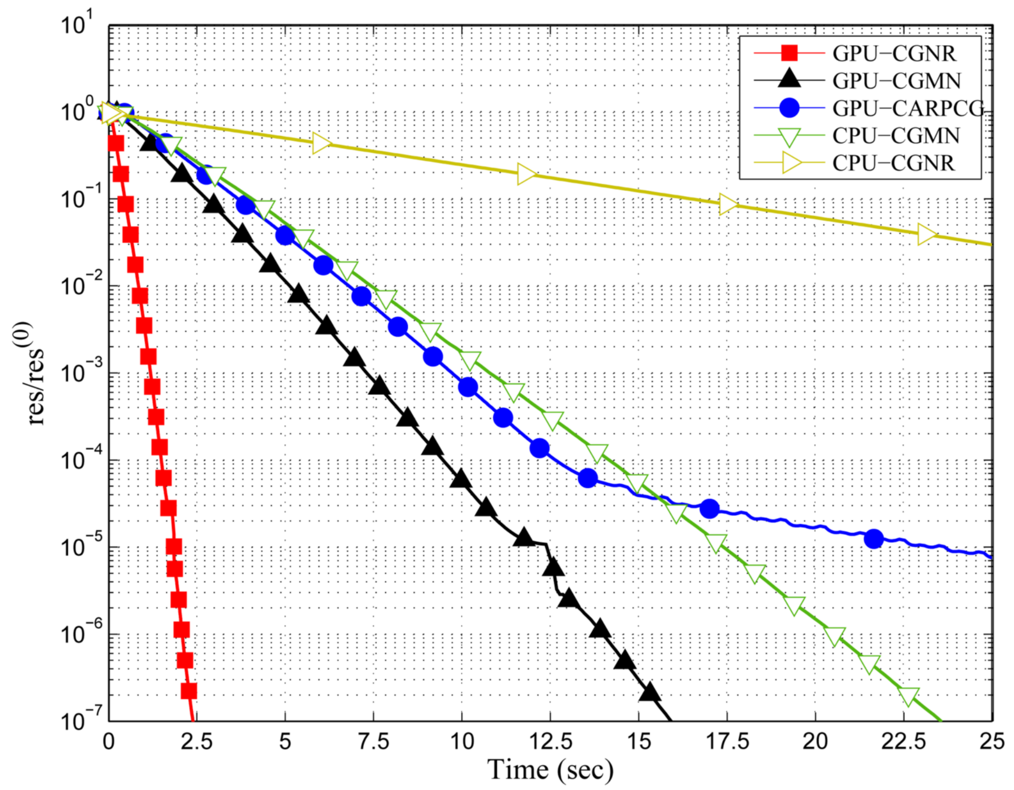

Figure 2.
Convergence results for Problem 2.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

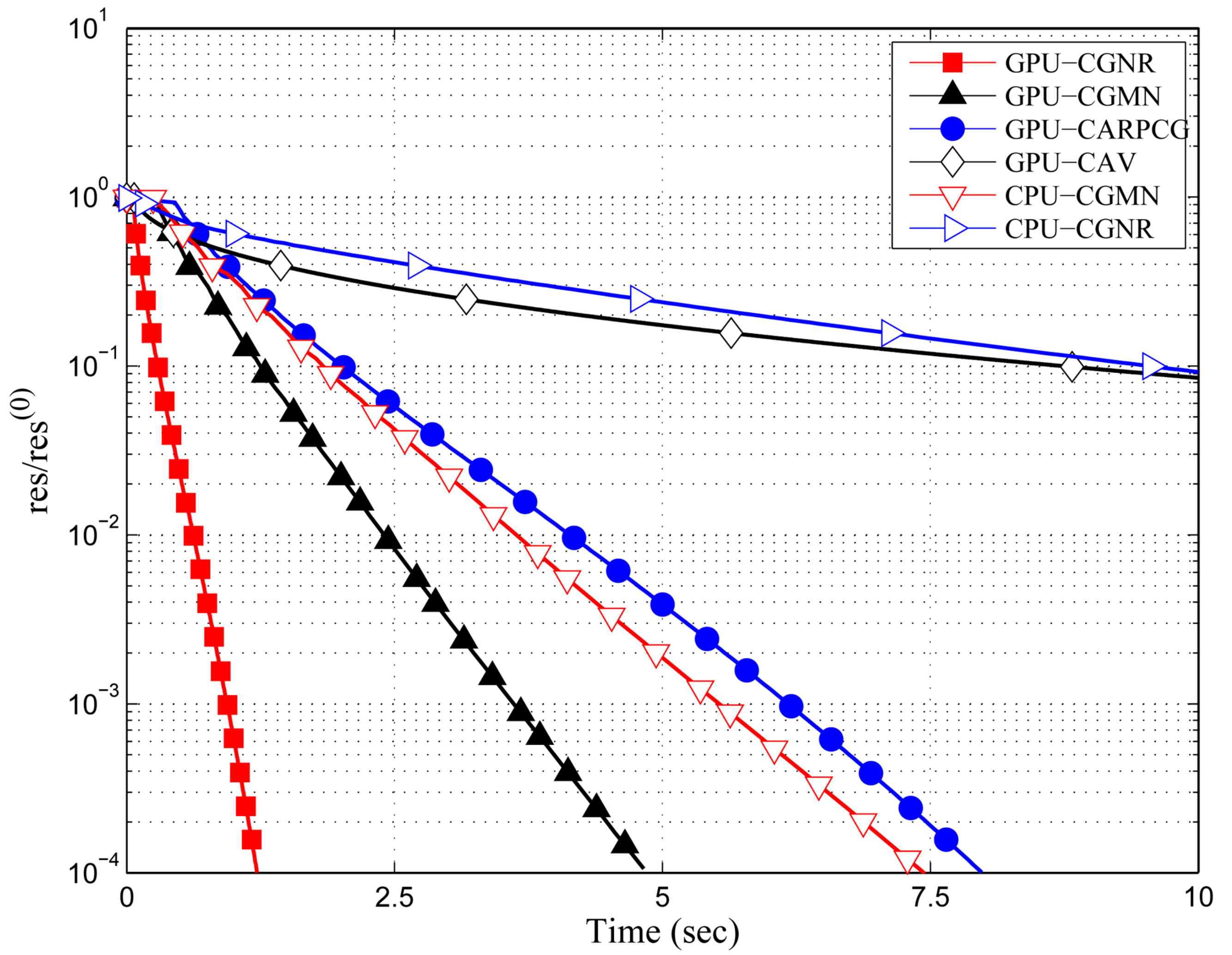

Figure 3.
Convergence results for Problem 3.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

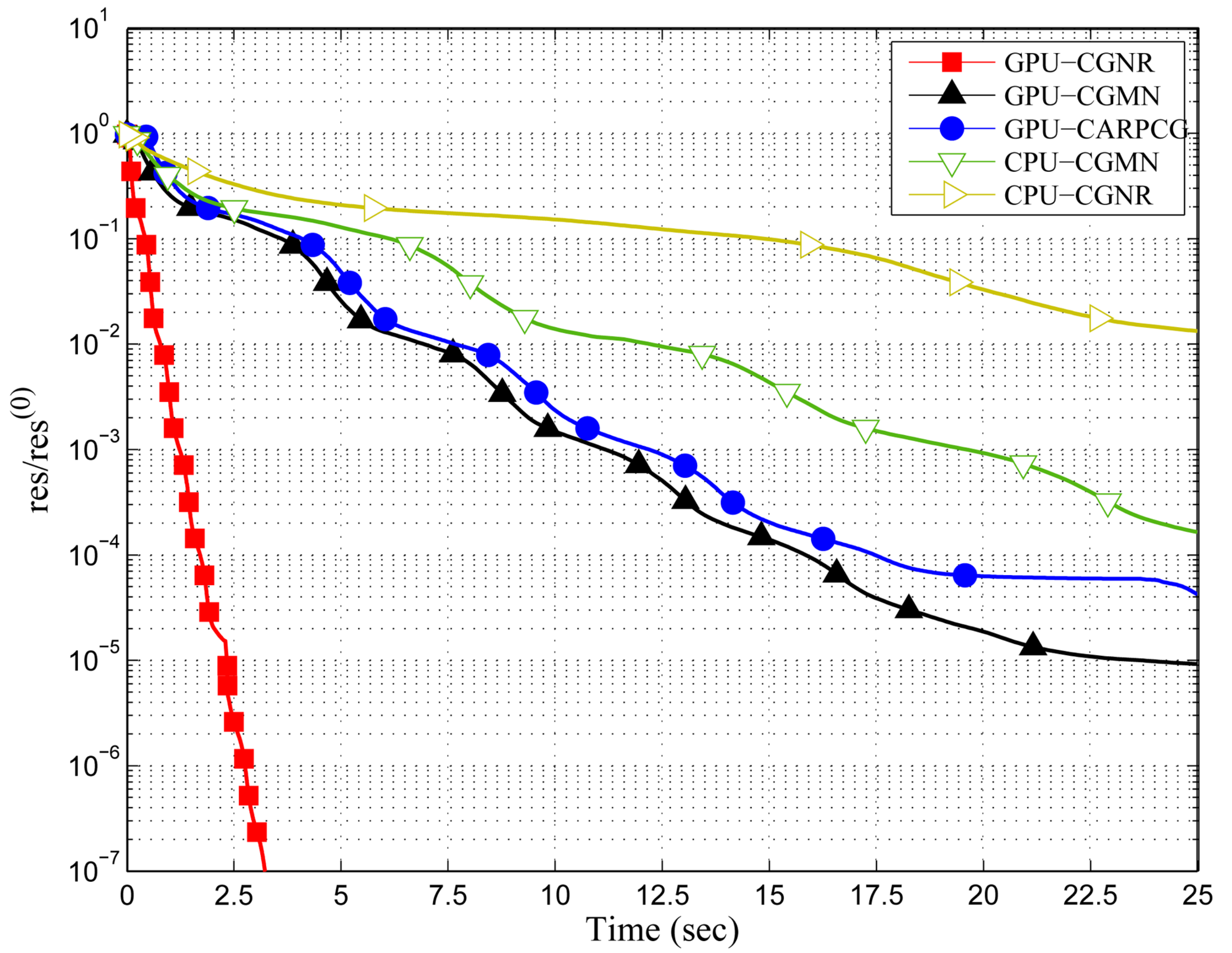

Figure 4.
Convergence results for Problem 4.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

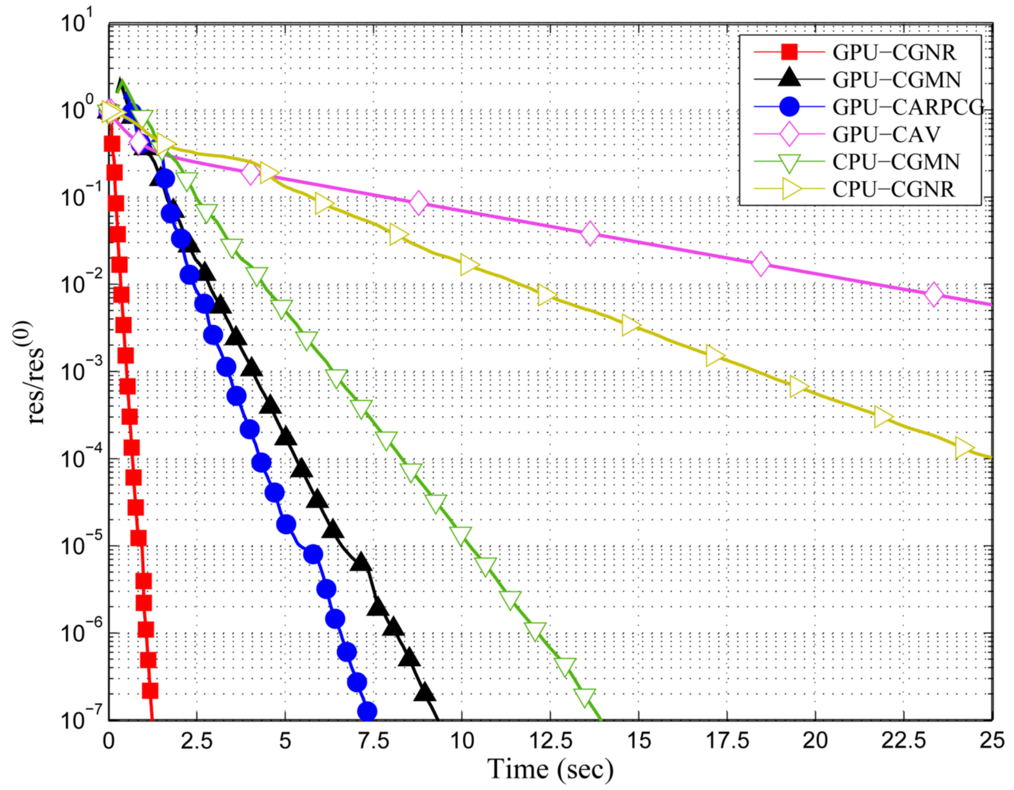

Figure 5.

Convergence results for Problem 5.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

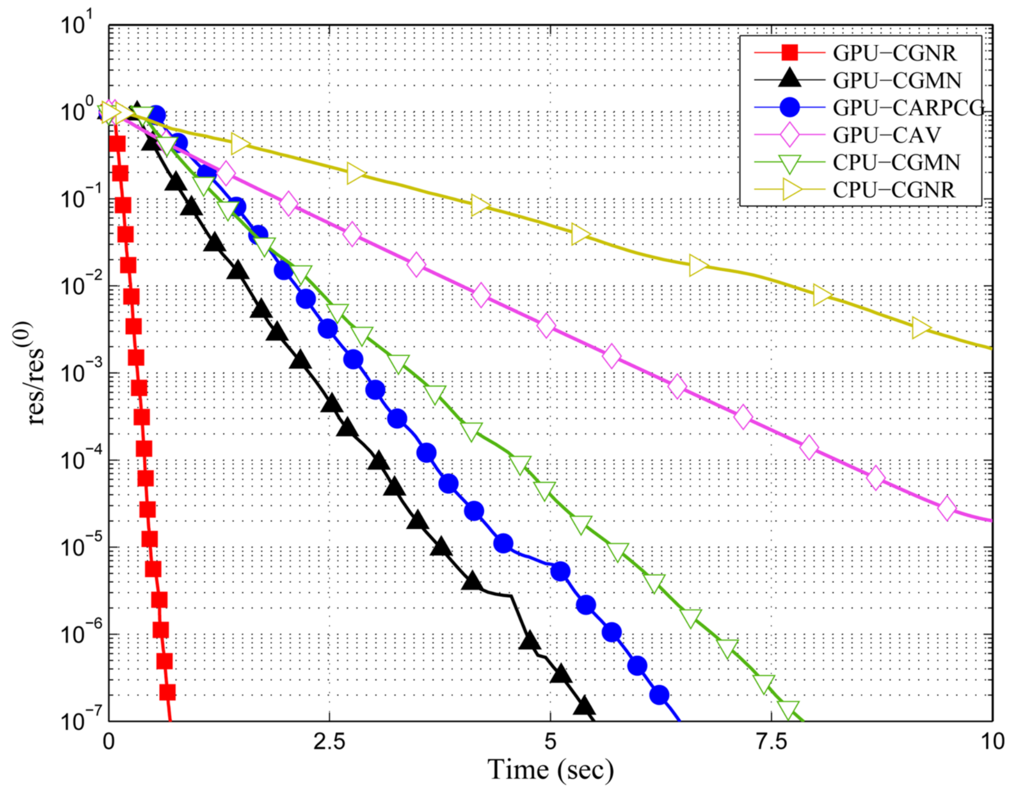

Figure 6.
Convergence results for Problem 6.

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

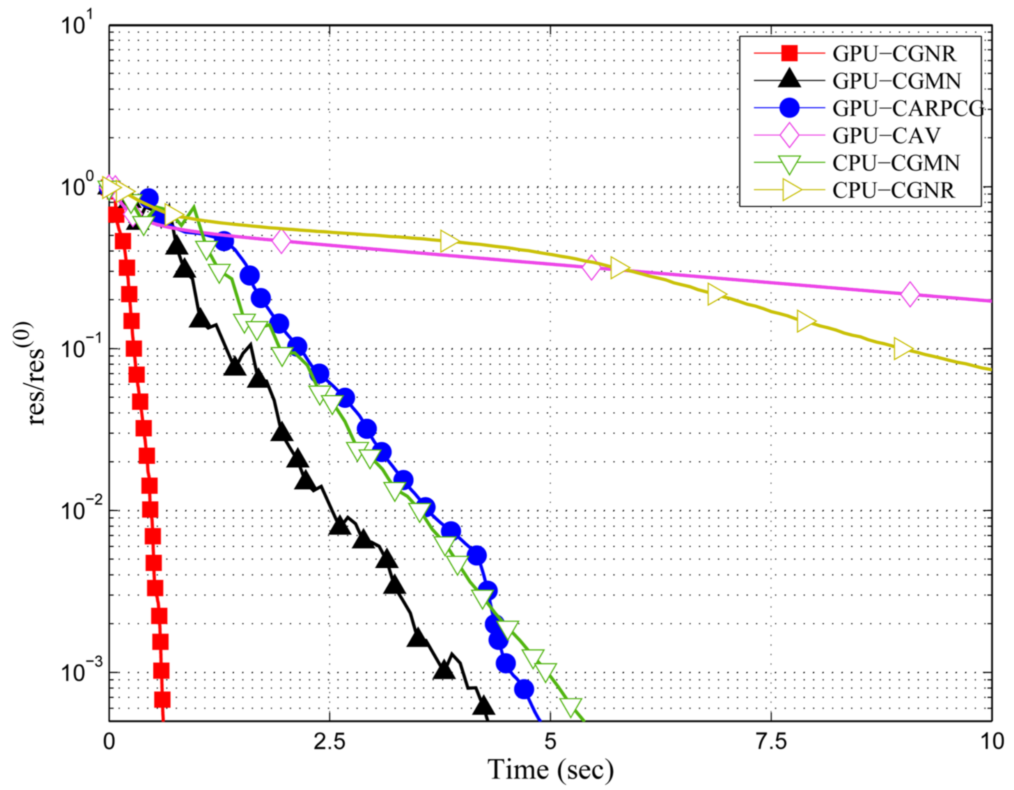

Figure 7.
Convergence results for Problem 7.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

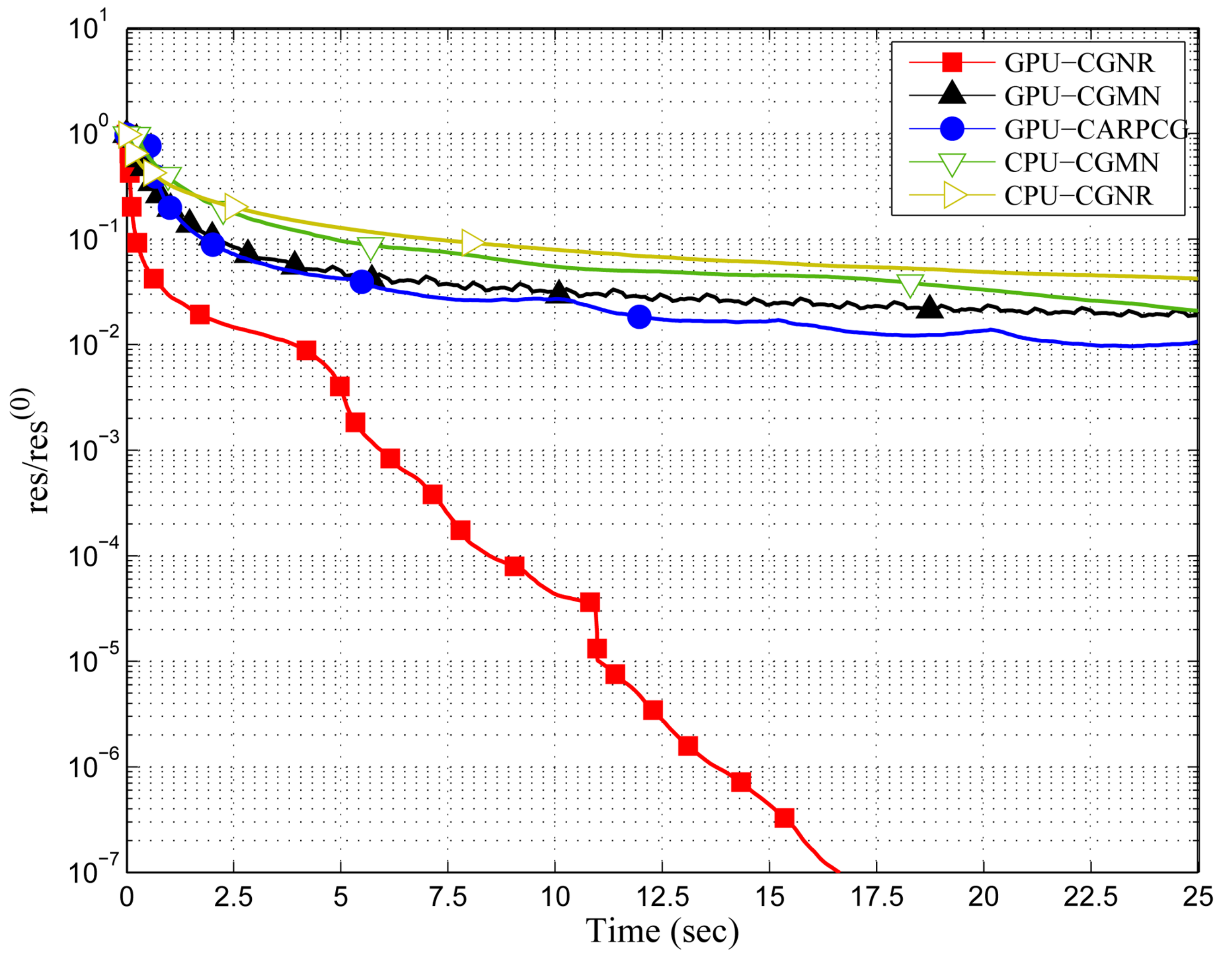

Figure 8.

Convergence results for Problem 8.

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

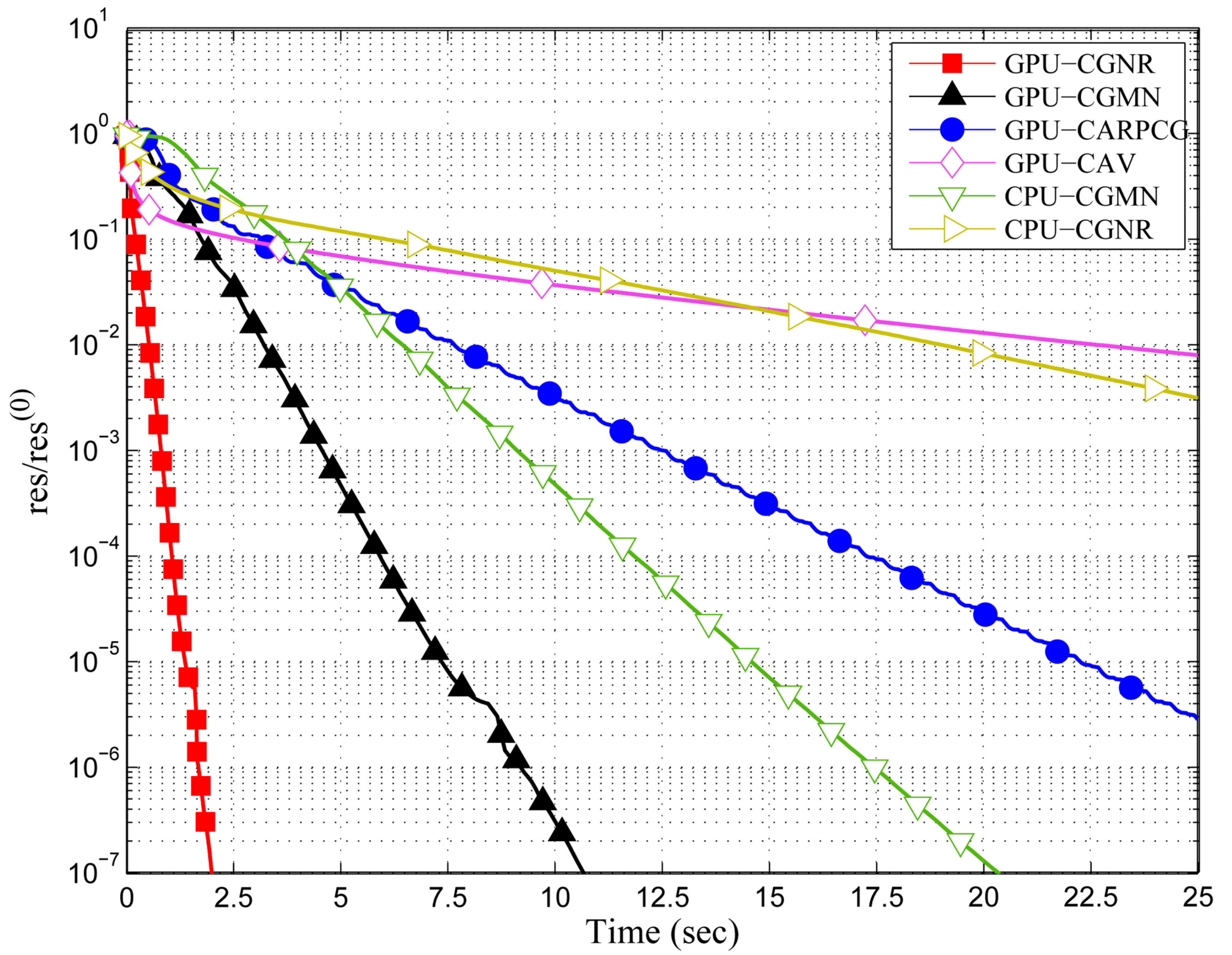

Figure 9.
Convergence results for Problem 9.

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

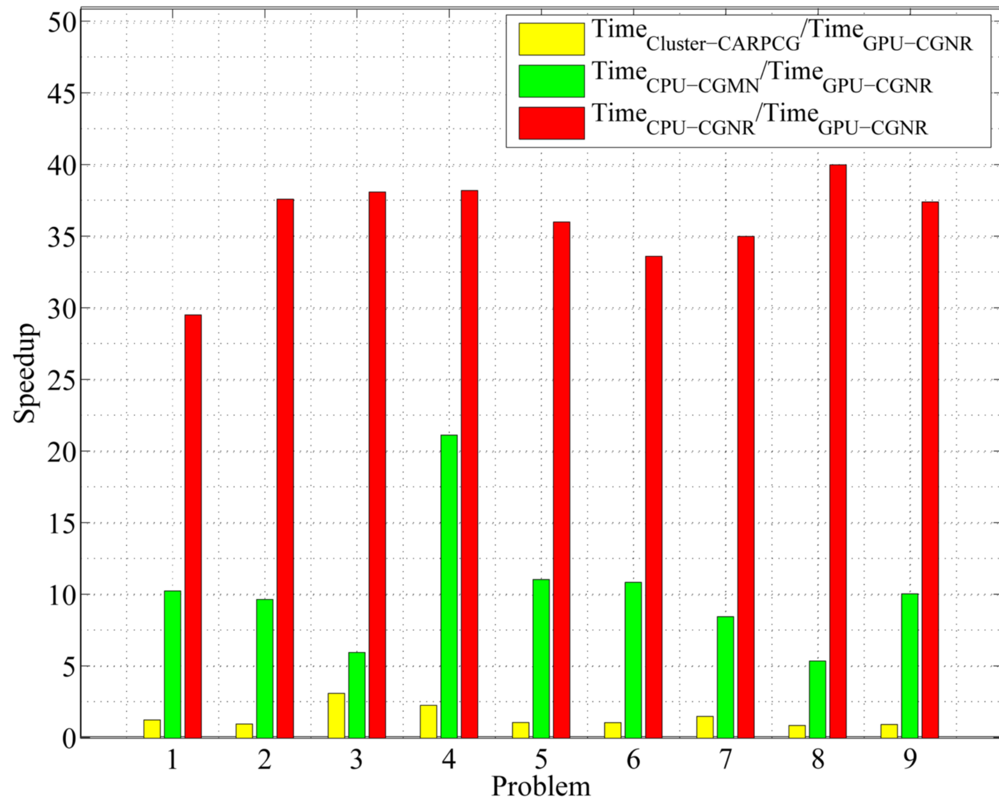

Figure 10.
GPU speedups in comparison to the CPU CGMN and the 16-node cluster.

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 1

Run times of Kaczmarz's algorithm for dense $n \times n$ systems

	GPU time (sec)		CPU time (sec)
Problem size (*n*)	**Memory initialization**	**Per iteration**	**Per iteration**
1000	0.003	0.034	0.014
2500	0.016	0.036	0.020
5000	0.061	0.039	0.034
7500	0.137	0.044	0.050
10000	0.245	0.043	0.064
15000	0.547	0.055	0.102

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 2

Run times of Cimmino's algorithm for dense $n \times n$ systems

	GPU time (sec)		CPU time (sec)
Problem size (*n*)	**Memory initialization**	**Per iteration**	**Per iteration**
1000	0.004	0.624	3.121
2500	0.018	5.547	18.702
5000	0.063	15.496	90.354
10000	0.265	28.634	349.058

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 3

Run times of CAV for $n \times n$ banded systems

	GPU time (sec)		CPU time (sec)
Problem size (n)	**Memory initialization**	**Per iteration**	**Per iteration**
1000	0.001	0.113	0.289
2500	0.001	0.188	0.657
5000	0.003	0.345	1.531
10000	0.005	0.720	3.145
15000	0.007	0.933	4.396
20000	0.008	1.060	5.021
25000	0.009	1.382	6.268
30000	0.011	1.614	7.515

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

Table 4

Run times of CAV for banded 25, 000 × 25, 000 systems

	GPU time (sec)		CPU time (sec)
Band size	**Memory initialization**	**Per iteration**	**Per iteration**
1	0.002	0.151	1.955
2	0.006	0.199	1.950
4	0.006	0.296	1.815
8	0.008	0.492	2.718
16	0.009	1.249	4.417
32	0.011	1.844	7.729
64	0.014	5.168	14.630

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 5

Average GFLOPS performance over the nine problems. For GPU-CAV, the average is over the six problems that it does not fail

Algorithm	GFLOPS
GPU-CGNR	11.82
GPU-CGMN	0.32
GPU-CARPCG	0.66
GPU-CAV	15.26
CPU-CGMN	0.2
CPU-CGNR	0.31

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 6

Number of iterations for Problems 1–9

	GPU				CPU		Cluster [10]
Problem	**CGNR**	**CGMN**	**CARP-CG**	**CAV**	**CGNR**	**CGMN**	**CARP-CG**
1	389	85	157	97,559	318	74	97
2	1023	177	1169	–	991	169	176
3	514	54	184	11,699	513	54	282
4	1398	518	1077	–	1382	487	578
5	506	103	165	81,024	499	99	105
6	268	60	144	13,897	262	57	62
7	247	53	106	59,728	247	38	77
8	7404	1117	2341	–	6418	636	1121
9	848	118	628	131,051	827	142	142

–: Algorithm failed

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 7

Runtimes (sec) for Problems 1–9

	GPU				CPU		Cluster [10]
Problem	**CGNR**	**CGMN**	**CARP-CG**	**CAV**	**CGNR**	**CGMN**	**CARP-CG**
1	1.01	7.79	6.61	110.54	29.92	10.40	1.3
2	2.43	15.95	56.85	–	91.52	23.63	2.45
3	1.26	4.92	8.03	97.15	48.13	7.56	3.95
4	3.26	46.29	45.11	–	124.79	69.24	7.5
5	1.27	9.40	7.45	91.91	45.80	14.04	1.4
6	0.73	5.56	6.48	15.84	24.59	7.97	0.8
7	0.65	4.34	4.91	67.76	22.82	5.52	1.0
8	16.68	220.06	115.36	–	668.38	89.91	15.0
9	2.02	10.70	32.31	149.56	75.78	20.47	1.95

–: Algorithm failed

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 8

Ratio of iterative-refinement iterations over double-precision iterations for each method

Problem	CGNR	CGMN	CARP-CG
1	1.22	1.14	1.61
2	1.03	1.04	6.53
3	1.00	1.00	0.65
4	1.01	1.06	1.86
5	1.01	1.04	1.57
6	1.02	1.05	2.32
7	1.00	1.39	1.37
8	1.15	1.75	2.08
9	1.02	0.83	4.42

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Table 9

Number of iterative refinements for Problems 1–9

Problem	CGNR	CGMN	CARP-CG	CAV
1	2	2	2	4
2	2	2	78	–
3	1	1	1	9
4	2	2	2	–
5	5	2	3	9
6	2	2	2	2
7	1	4	2	6
8	2	99	23	–
9	2	2	57	14

–: Algorithm failed

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

Algorithm 1

Kaczmarz's algorithm

1 **for** $k \leftarrow 0$ until convergence or maximum number of iterations met **do**
2 $\quad i \leftarrow k \bmod m + 1$
3 $\quad s^{(k)} \leftarrow \lambda_i \frac{b_i - \langle a_i, x^{(k)} \rangle}{\|a_i\|^2} a_i$
4 $\quad x^{(k)} \leftarrow x^{(k)} + s^{(k)}$
5 **end**

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Algorithm 2

Cimmino's algorithm

1 $D \leftarrow \frac{1}{m}\text{diag}\left(\frac{1}{\|a_1\|^2}, \frac{1}{\|a_2\|^2}, \ldots, \frac{1}{\|a_m\|^2}\right)$
2 **for** $k \leftarrow 0$ until convergence or maximum number of iterations met **do**
3 | $x^{(k+1)} \leftarrow x^{(k)} + \lambda_k A^T D \left(b - Ax^{(k)}\right)$
4 **end**

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

Algorithm 3

Component-averaging (CAV)

---

1 Let s_j denote the number of nonzeros in column j
2 $S \leftarrow \text{diag}\left(s_1, s_2, \ldots, s_n\right)$
3 $D_S \leftarrow \text{diag}\left(\frac{1}{\|a_1\|_S^2}, \frac{1}{\|a_2\|_S^2}, \ldots, \frac{1}{\|a_m\|_S^2}\right)$
4 **for** $k \leftarrow 0$ until convergence or maximum number of iterations met **do**
5 | $x^{(k+1)} \leftarrow x^{(k)} + \lambda_k A^T D_S \left(b - Ax^{(k)}\right)$
6 **end**

NIH-PA Author Manuscript

NIH-PA Author Manuscript

NIH-PA Author Manuscript

Algorithm 4

Conjugate gradient normal residual (CGNR)

---

1 $r^{(0)} \leftarrow b - Ax^{(0)}$
2 $z^{(0)} \leftarrow A^T r^{(0)}$
3 $p^{(0)} \leftarrow z^{(0)}$
4 **for** $i \leftarrow 0$ until convergence or maximum number of iterations met **do**
5 $\quad w^{(i)} \leftarrow Ap^{(i)}$
6 $\quad \alpha^{(i)} \leftarrow \left\|z^{(i)}\right\|^2 / \left\|w^{(i)}\right\|^2$
7 $\quad x^{(i+1)} \leftarrow x^{(i)} + \alpha^{(i)} p^{(i)}$
8 $\quad r^{(i+1)} \leftarrow r^{(i)} - \alpha^{(i)} w^{(i)}$
9 $\quad z^{(i+1)} \leftarrow A^T r^{(i+1)}$
10 $\quad \beta^{(i)} \leftarrow \left\|z^{(i+1)}\right\|^2 / \left\|z^{(i)}\right\|^2$
11 $\quad p^{(i+1)} \leftarrow z^{(i+1)} + \beta^{(i)} p^{(i)}$
12 **end**

NIH-PA Author Manuscript NIH-PA Author Manuscript NIH-PA Author Manuscript

Algorithm 5

Symmetric successive overrelaxation (SSOR)-preconditioned conjugate gradient (CGMN)

1 $t \leftarrow x^{(0)}$
2 **for** $i \leftarrow 1$ **to** $2m$ **do**
3     $j \leftarrow \min(i, 2m + 1 - i)$
4     $t \leftarrow t + \lambda \frac{b_j - a_j^T t}{\|a_j\|^2} a_j$
5 **end**
6 $p^{(0)} \leftarrow t - x^{(0)}$
7 $r^{(0)} \leftarrow p^{(0)}$
8 **for** $k \leftarrow 0$ until convergence or maximum number of iterations met **do**
9     $t \leftarrow p^{(k)}$
10   **for** $i \leftarrow 1$ **to** $2m$ **do**
11         $j \leftarrow \min(i, 2m + 1 - i)$
12         $t \leftarrow t + \lambda \frac{a_j^T t}{\|a_j\|^2} a_j$
13   **end**
14     $q^{(k)} \leftarrow p^{(k)} - t$
15     $\alpha^{(k)} \leftarrow \left\|r^{(k)}\right\|^2 / \left\langle p^{(k)}, q^{(k)} \right\rangle$
16     $x^{(k+1)} \leftarrow x^{(k)} + \alpha^{(k)} p^{(k)}$
17     $r^{(k+1)} \leftarrow r^{(k)} - \alpha^{(k)} q^{(k)}$
18     $\beta^{(k)} \leftarrow \left\|r^{(k+1)}\right\|^2 / \left\|r^{(k)}\right\|^2$
19     $p^{(k+1)} \leftarrow r^{(k+1)} + \beta^{(k)} p^{(k)}$
20 **end**

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript

Algorithm 6

Conjugate gradient accelerated component-averaged row projections (CARP-CG)

```
1  t ← x^(0)
2  Perform the following Kaczmarz double sweep using CARP
3  for i ← 1 to 2m do
4  |   j ← min(i, 2m + 1 − i)
5  |   t ← t + λ (b_j − a_j^T t) / ||a_j||^2 a_j
6  end
7  p^(0) ← t − x^(0)
8  r^(0) ← p^(0)
9  for k ← 0 until convergence or maximum number of iterations met do
10 |   t ← p^(k)
11 |   Perform the following Kaczmarz double sweep using CARP
12 |   for i ← 1 to 2m do
13 |   |   j ← min(i, 2m + 1 − i)
14 |   |   t ← t + λ (a_j^T t) / ||a_j||^2 a_j
15 |   end
16 |   q^(k) ← p^(k) − t
17 |   α^(k) ← ||r^(k)||^2 / ⟨p^(k), q^(k)⟩
18 |   x^(k+1) ← x^(k) + α^(k) p^(k)
19 |   r^(k+1) ← r^(k) − α^(k) q^(k)
20 |   β^(k) ← ||r^(k+1)||^2 / ||r^(k)||^2
21 |   p^(k+1) ← r^(k+1) + β^(k) p^(k)
22 end
```

Algorithm 7

CPU-GPU iterative refinement

```
1  for i ← 0 to (Maximum-CPU-Iterations−1) do
2  |   Compute d = b − Ax in double precision on the CPU
3  |   Transfer d to the GPU in single precision
4  |   Solve Ac = d approximately on the GPU
5  |   ||c^(0)|| = Maximum Representable Number
6  |   for k ← 1 to (Maximum-GPU-Iterations−1) do
7  |   |   Carry out one iteration of CGNR, CGMN, CARP-CG, or CAV
   |   |   on the GPU
8  |   |   If ||c^(k)|| / ||c^(k−1)|| > 1.0 terminate loop
9  |   end
10 |   Transfer c to the CPU
11 |   Update x = x + c in double precision on the CPU
12 |   If res^(i)/res^(0) < TOL, terminate; else, iterate
13 end
```

NIH-PA Author Manuscript
NIH-PA Author Manuscript
NIH-PA Author Manuscript