Parallel Sparse Approximate Inverse Preconditioning on Graphic Processing Units

Maryam Mehri Dehnavi, *Member*, *IEEE*, David M. Fernández, *Member*, *IEEE*, Jean-Luc Gaudiot, *Fellow*, *IEEE*, and Dennis D. Giannacopoulos, *Member*, *IEEE*

Abstract—Accelerating numerical algorithms for solving sparse linear systems on parallel architectures has attracted the attention of many researchers due to their applicability to many engineering and scientific problems. The solution of sparse systems often dominates the overall execution time of such problems and is mainly solved by iterative methods. Preconditioners are used to accelerate the convergence rate of these solvers and reduce the total execution time. Sparse approximate inverse (SAI) preconditioners are a popular class of preconditioners designed to improve the condition number of large sparse matrices. We propose a GPU accelerated SAI preconditioning technique called GSAI, which parallelizes the computation of this preconditioner on NVIDIA graphic cards. The preconditioner is then used to enhance the convergence rate of the BiConjugate Gradient Stabilized (BiCGStab) iterative solver on the GPU. The SAI preconditioner is generated on average 28 and 23 times faster on the NVIDIA GTX480 and TESLA M2070 graphic cards, respectively, compared to *ParaSails* (a popular implementation of SAI preconditioners on CPU) single processor/core results. The proposed GSAI technique computes the SAI preconditioner in approximately the same time as *ParaSails* generates the same preconditioner on 16 AMD Opteron 252 processors.

Index Terms—Numerical algorithms, parallel algorithms, graphics processors, parallel programming, conditioning

1 Introduction

Mathematical physics and engineering problems in a broad range of applications have grown larger and more complex in the past few decades leading to large scale simulations. These simulations generally involve the use of techniques such as the finite element method and the finite difference time domain method which are used to discretize, assemble and solve such systems [1], [2]. One of the most time consuming steps in the aforementioned techniques is solving the system of equations proceeding the systems assembly stage. The solution of such systems is achieved by either direct or iterative methods. For larger and sparser systems, direct methods often suffer from high computational complexity and are notoriously difficult to implement in parallel due to their recursive nature [3]. A more viable alternative to solving large linear systems is using iterative solvers. Krylov methods are a popular class of these solvers with techniques such as the generalized minimum RESidual (GMRES) and BiConjugate Gradient Stabilized (BiCGStab) [1]. Krylov solvers generally involve less computations and memory requirements compared to direct methods but suffer from slow convergence rates especially for ill-conditioned matrices [4]. Because of their slow convergence these methods are frequently used with preconditioners.

Preconditioners are designed to accelerate the convergence rate of iterative solvers for a majority of applications. Applying the preconditioner M, to both sides of the linear systems equation $Ax = b$, reduces the number of iterations and accelerates the execution time of the solver. A popular class of preconditioners suitable for parallelization and efficient for a large class of problems are the sparse approximate inverse (SAI) preconditioners. Although computing SAI preconditioners is generally expensive on a single processor, constructing them on parallel architecture is relatively fast. By generating a denser preconditioner, SAI preconditioning can reduce iterations in iterative solvers considerably and be applied to a broad range of applications. Previous work has accelerated the computation of this preconditioner on multiple processors [5], [6], [7], [8], [9], [10], [11], [12], [13] as well as multicore [14], [15] and many-core architecture [16].

Graphic processing units have become an important resource for scientific computing in recent years [17]. With easy to learn application programming interfaces such as compute unified device architecture (CUDA) [18] introduced by NVIDIA, general purpose programming for modern scientific computations on GPUs gained considerable attention. Using a single data multiple thread paradigm, GPU threads grouped into thread blocks run compute intensive parts of an application in parallel. The GPU has an on-board global memory with long access latency, a fast access shared memory, registers and caches. Every 32 threads in a thread block execute the same instruction and are called a warp. In this paper, we present a new GPU accelerated SAI preconditioning technique called GSAI, which parallelizes the computation of SAI preconditioners on NVIDIA GPUs. Major contributions of the proposed GSAI technique are as follows:

- *M.M. Dehnavi is with the Massachusetts Institute of Technology, G770, 32 Vasser Street, Cambridge, MA 02139. E-mail: mmehri@mit.edu.*
- *D.M. Fernández and D.D. Giannacopoulos are with the McGill University, McConnell Engineering Building, 3480 rue University Montreal, Quebec, Canada H3A 0E9. E-mail: davidmoises@gmail.com, dennis.giannacopoulos@mcgill.ca.*
- *J.-L. Gaudiot is with the Department of Electrical Engineering and Computer Science, University of California, Irvine, Irvine, CA 92697-2625. E-mail: gaudiot@uci.edu.*

Manuscript received 6 June 2011; revised 23 July 2012; accepted 26 Sept. 2012; published online 3 Oct. 2012.
Recommended for acceptance by S. Ranka.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-06-0362.
Digital Object Identifier no. 10.1109/TPDS.2012.286.

- The preconditioner M is generated in parallel on the GPU; each GPU warp computes one column of M.
- Large data structures are stored in GPU global memory and memory space is reused by dividing the computation of M between many GPU kernels.
- Memory accesses, vector multiplications, inner products, *QR* decomposition, and triangular solves are computed in parallel inside a GPU warp.
- The preconditioner is assembled in a compressed storage format and then applied to the BiCGStab iterative solver, which is also accelerated on the GPU.

2 SAI Preconditioning

A sparse approximate inverse preconditioner approximates the inverse of A using a sparse matrix M to improve the condition number of the linear system of equations $Ax = b$. M is computed using the least-squares methods and by minimizing the matrix residual norm

$$\|AM - I\|_F^2. \tag{1}$$

The above equation is then separated into n independent least square problems

$$\min_{m_k} \|Am_k - e_k\|_2^2, \quad k = 1, 2, \ldots, n, \tag{2}$$

where e_k is the kth column of the identity matrix and m_k represents column k in matrix M. The degrees of freedom in solving the above equations are the locations and values of the nonzeros in M. Based on the degree of freedom used, sparse approximate inverse preconditioner generation is classified as adaptive or static (a priori). In adaptive schemes ([9], [19], [20], etc.) the sparsity of M is initially set to a simple pattern such as diagonal, this pattern is then augmented until a threshold on the residual norm or a maximum on the number of nonzeros in M is reached. Although adaptive methods have broadened the scope of problems which can be solved using SAI preconditioning, by utilizing additional degrees of freedom in minimizing (2), the preconditioner generation becomes generally very expensive requiring many reruns to determine the appropriate values of various parameters involved, such as tolerance [21], maximum improvements per step [21], number of nonzeros per step [21], and so on for each problem. On the other hand, static preconditioning ([6], [12], [21], [22], [23]) determines the sparsity of M in a preprocessing step limiting the degrees of freedom in (1) to the nonzero values of M.

Previous work has introduced various techniques to determine a more accurate approximation of M prior to computing the preconditioner and have shown that static schemes are more efficient than adaptive techniques in improving the condition number of the A matrix if the sparsity of M is better approximated. Since the focus of this work is not to introduce a better initial guess for the M preconditioner but to accelerate the computation of M (1), for general static (a priori) SAI preconditioners, we use the most popular approximate of M which is based on sparsifications [24] of A. $M(i, j)$ is considered a nonzero if the condition

i) J is constructed based on (3).
j) Columns of A in J are selected and matched to construct I.
k) $\hat{A}$ is constructed and decomposed using QR Gram-Schmidt.
l) Values in $\hat{m}_k$ are computed using $\hat{m}_k = R^{-1}Q^T\hat{e}_k$ and scattered back to M.

Fig. 1. Steps involved in constructing static sparse approximate inverse preconditioners.

$$|A(i,j)| > (1 - \tau)\max_j |A(i,j)|, \quad 0 \leq \tau \leq 1, \tag{3}$$

is satisfied, where τ is a user defined tolerance parameter (the main diagonal is always included). Based on (3), for smaller τ parameters more nonzeros entries in A are dropped resulting in a sparser preconditioner; for τ equal to 1 the sparsity pattern assumed for M would be the same as the sparsity of A. If a more accurate approximate of the sparsity of M is known for a specific application it can be used instead of (3). Knowing the sparsity of M before solving (1), reduces (2) to

$$\min_{\hat{m}_k} \|\hat{A}\hat{m}_k - \hat{e}_k\|_2^2, \quad k = 1, 2, \ldots, n, \tag{4}$$

$\hat{m}_k$ is the reduced vector of unknows $m_k(J)$, where J is the set of indices j such that $m_k(j) \neq 0$. Considering I as a set of indices i such that $A(i, J)$ is not zero, $\hat{A}$ is the submatrix $A(I, J)$ where all zero rows in $A(., J)$ are deleted. The dimension of $\hat{A}$ is equal to $n_1 \times n_2$ where n_1 and n_2 are the number of indices in I and J, respectively. Finally, $\hat{e}_k$ represents $e_k(I)$. To construct and solve (4) for each column k of M, the steps in Fig. 1 should be computed for each k (more information on the above implementations and the steps in Fig. 1 can be found in previous work on SAI preconditioners specifically [6], [7], [9], [11], [21]).

Factorized sparse approximate inverse preconditioners are another class of SAI preconditioning techniques developed in [25], [26], [27], [28], [29], [30], [31]. This class of preconditioners are less popular than the kind based on Frobenius norm minimization (1) [4] and can fail due to breakdowns during an incomplete factorization process. A comparative study of various SAI preconditioners is presented in [32]. Sparsification is a method used to diminish the pattern of A when it is relatively full and generate a sparser preconditioner and can be implemented in both adaptive and static SAI preconditioner construction algorithms. Initially introduced by Kolotilina [24] for computing SAI preconditioners for dense matrices, sparsification is also used by Tang [33] to enhance the condition number of anisotropic problems. Costgrov et al. [34] also propose augmenting the pattern of A for constructing sparse approximate inverse preconditioners. SAI preconditioner proposed by All'eon et al. [35] and *ParaSails* [7] introduced by Chow [6] use a priori sparsity patterns based on powers of sparsified matrices for partial differential equation (PDE) problems. Sparsification is also implemented in *SPAI 3.2* [21] by eliminating small values in A before computing the preconditioner. The equation used in the proposed GSAI technique (3) also allows for sparsifying A using a tolerance parameter τ. Applying sparsification to the preconditioner after it has been produced is also studied

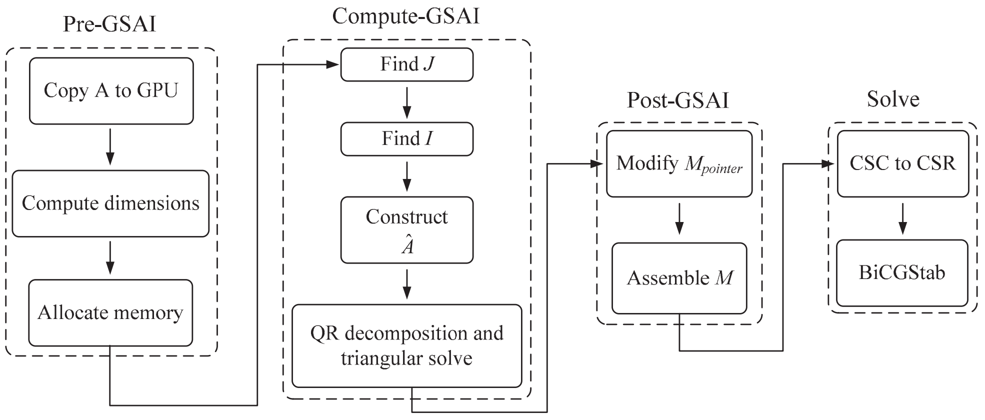


Fig. 2. The four stages in implementing SAI preconditioners using GSAI on NVIDIA GPUs.

in [25] and [36]. If an effective sparsification is known for a specific problem it can be added to the *Pre-GSAI* stage (see Fig. 2) in the GSAI method proposed.

Most of the work on SAI preconditioners presents techniques to parallelize the computation of the preconditioner on multiprocessor architectures [5], [6], [7], [8], [9], [10], [11], [12], [13], by distributing the computation of the columns in M between multiple processors. Techniques such as grouping communications [11], dictionary-based methods [8] and latency-tolerant hybrid SAI preconditioning [10] are proposed in these work, to further enhance the execution time of SAI preconditioners on multiprocessors. *ParaSails* [7] and *SPAI 3.2* [21] are two of the most popular open source implementations of the sparse approximate inverse preconditioner on single- and multiprocessor platforms and are used for comparison in a majority of previous work [4], [6], [8], [10]. While *ParaSails* uses a priori approximation of M to generate the preconditioner, both adaptive and static SAI preconditioners are implemented in *SPAI 3.2*. Similar to *SPAI 3.2* the preconditioned problem in GSAI is solved using the BiCGStab iterative solver (*ParaSails* implements the GMRES and CG iterative solvers). Chow [6] compares the performance of *ParaSails* to *SPAI 3.2* and shows *ParaSails* generates the SAI preconditioner considerably faster than *SPAI 3.2*. We compare the GSAI preconditioner generation time on GPUs to *ParaSails* on single- and multiprocessor platforms.

Although parallelizing sparse approximate inverse preconditioners on more than one processor has been extensively studied in previous work which succeeded to enhance the execution speed of such preconditioners considerably, few works have studied the possibility of accelerating these preconditioners on multi/many core architectures. Gravvanis et al. [14], [15] attempt to accelerate a SAI preconditioned BiCGStab iterative solver on Intel multicore architecture by allocating the computation of each iteration of the iterative solver to a different thread; implementation details on how to accelerate the preconditioner computation on a multicore are not presented in this work. Xu et al. [16] accelerate factorized SAI on NVIDIA GPUs. The paper mainly describes how to accelerate the sparse matrix vector multiplication kernel (SpMV) in the iterative solver but details for computing the SAI preconditioner have not been presented (other accelerations of the SpMV kernel are presented in [37], and [38] and CUSPARSE [41]).

3 Parallel SAI on NVIDIA GPUs

The SAI preconditioner is computed in parallel on GPUs by allocating the computation of each column of M to one warp. Accelerating the SAI preconditioner involves local (per warp) parallelization of various computing kernels such as QR decomposition, dot products, sorting vector values, finding the maximum value in a vector, and so on. One of the major challenges in computing SAI preconditioners on GPUs is the limited size of global and shared memory and the generation of large data structures. Proposing techniques to reuse memory space and minimize the allocated memory to data structures in the kernel are key factors in producing SAI preconditioners for large problems on GPUs. In the following implementation details to overcome the above constraints and implement in parallel the computing kernels involved in solving $Ax = b$ using SAI preconditioners are presented.

Computing the SAI preconditioner in parallel on GPUs involves the implementation of steps introduced in Fig. 1, which we implemented in a stage called *Compute-GSAI* (see Fig. 2). In this stage, every 32 threads (one warp) on the GPU computes one column of M (m_k) by executing the steps in Fig. 1. Each warp first finds the dimensions of its corresponding $\hat{A}$ matrix (4) and assembles it. The local $\hat{A}$ matrices, which are very small compared to A, are then decomposed (local decompositions per warp for each $\hat{A}$) using the Gram Schmidt method [1] and m_k is computed. SAI preconditioning on GPUs requires two additional steps (*Pre-GSAI* and *Post-GSAI*) which handle GPU memory allocation, define required data structures, gather results and determine the required number of kernel (hereafter *kernel* refers to a CUDA *kernel*) calls based on the problem size and available GPU memory. Thus, solving the $Ax = b$ linear systems equations on the GPU using SAI preconditioners consists of four major steps (see Fig. 2):

1. *Pre-GSAI:* involves reading A in a compressed sparse format [39] and transferring it to GPU, allocating GPU memory space to the preconditioner M and other data structures and determining the number of kernel calls based on the available global memory.

2. *Compute-GSAI:* computes the SAI preconditioner on the GPU and scatters the produced columns back to M on GPU global memory.
3. *Post-GSAI:* revises the assigned global memory space to M by releasing extra memory space allocated to M and assembles M on the GPU in compressed column storage (CSC) [39] format.
4. *Solver:* converts both M and A from CSC to CSR (compressed row storage [39]) on the GPU to accelerate the iterative solver execution time and solves $Ax = b$ using the computed SAI preconditioner and the BiCGStab iterative solver.

The rest of this section is organized as follows. Section 3.1 introduces implementation details of the above steps and the kernel/function calls involved in each stage. Managing global and shared memory, determining the amount of memory required for each data structure and deciding the necessary number of kernel calls are proposed in Section 3.2.

3.1 GSAI Steps

The proposed GSAI preconditioning method computes the SAI preconditioner on NVIDIA GPUs in three major steps, namely, *Pre-GSAI*, *Compute-GSAI*, and *Post-GSAI*, the generated preconditioner is then passed to the *Solver* stage (see Fig. 2) to precondition and solve the linear system.

3.1.1 Pre-GSAI Stage

Copy A to GPU. Sparse matrices are stored in memory using various compressed sparse storage formats such as CSR, CSC, and so on [39]. To compute the SAI preconditioner, the A matrix is initially stored in CSC format using three vectors called A_{value}, A_{index}, and $A_{pointer}$. The M matrix is also produced and stored in columns. A copy of the A matrix is transferred to GPU global memory.

Compute n_1 *and* n_2 *and allocate memory to M.* The preconditioner M is stored in global memory, thus memory should be allocated to M prior to the *Compute-GSAI* stage. Although the dimensions of M are the same as A it has to be stored in compressed format to fit on the GPU global memory. To reduce the amount of computation required to locate data structures used by each warp and regularize global memory accesses, equal memory space is allocated to each column of M using the *compute dimensions* kernel (see Fig. 2). The proposed memory allocation technique, introduces the need for the *Post-GSAI* step described in the next section, whose execution time is, however, negligible compared to *Compute-GSAI* as shown in the results section (see Section 4) and to the provided benefits. The kernel first finds the dimensions of local $\hat{A}$ matrices (n_1, n_2) and stores them on global memory and the maximum n_1 and n_2 values between all columns (called $n_{1,\max}$ and $n_{2,max}$) are then found. Since the number of nonzeros in the largest column of M is equal to $n_{2,max}$, global memory allocated to M would be equal to the number of columns in M multiplied by the number of bytes required to store $n_{2,max}$ floating point values (M_{value}). The row indices corresponding to the values of the preconditioner (M_{index}) and the number of nonzeros produced for each column of M ($M_{pointer}$) are stored in global memory. Besides allocating memory to the preconditioner M, the *allocate memory* step of the *Pre-GSAI* stage (see Fig. 2) assigns memory space to other data structures used during the computation of the SAI preconditioner (*Compute-GSAI*) and determines the number of kernel calls required to compute the SAI preconditioner. Details of these implementations are presented in Section 3.2 and Table 1.

TABLE 1
The Number of Elements in Each of the Data Structures Involved in GSAI and Their Size Based on Their Data Type

Data Structure	Number of elements	Type
A_{value}	non-zeros	double
A_{index}	non-zeros	integer
$A_{pointer}$	columns	integer
M_{value}	columns × $n_{2,max}$	double
M_{index}	columns × $n_{2,max}$	integer
$M_{pointer}$	columns	integer
$\hat{A}$ *(all columns)*	columns × $n_{1,max}$ × $n_{2,max}$	double
Q *(all columns)*	columns × $n_{1,max}$ × $n_{2,max}$	double
I_{index} *(all columns)*	columns × $n_{1,max}$	integer

Columns and nonzeros represent the number of columns and nonzeros in A.

3.1.2 Compute-GSAI Stage

To compute the SAI preconditioner on the GPU, the steps indicated in the *Compute-GSAI* stage in Fig. 2 have to be implemented in parallel on the GPU in a kernel called *compute preconditioner*. Each column of the preconditioner M is computed via one warp (32 threads in a block) and every block is assigned 256 threads (eight warps) to compute eight columns in parallel. The number of columns computed in one SM simultaneously will depend on the allocated shared memory per block and available resources per SM.

Find J. In this stage, the set J (the first step in Fig. 1) is constructed and loaded into a vector called J_{index}. Each warp in the kernel first loads the column in A corresponding to its index (the index is assigned to each warp based on the total number of warps launched on the GPU) and finds the largest element in the loaded column. The condition in (3) is then evaluated for each element of the loaded value vector simultaneously and the column index of elements satisfying the condition is stored in J_{index}.

Find I and construct the local $\hat{A}$. To determine I (see Fig. 1), the row indices of the first column referenced in J_{index} are first loaded into a vector called I_{index}. The row index vector of successive columns referenced by J_{index} are then loaded in order into shared memory and compared in parallel with values in I_{index}, new indices are tagged and later added to I_{index} to construct the set I. Local $\hat{A}$ matrices are constructed on global memory by loading columns indexed in J_{index} and matching them to I_{index} in parallel (see Fig. 3).

Local QR decomposition and triangular solves. Local QR decompositions are computed using the Gram Schmidt method [1], which was easier to parallelize inside a warp compared to other QR decomposition techniques. Each warp decomposes one $\hat{A}$ matrix, thus many QR decompositions are computed simultaneously via warps executing in parallel. Parallelism is also exploited in a warp by computing the local QR decompositions in parallel using the 32 threads inside a warp, for example, most of the operations in the QR decomposition technique such as memory loads and multiplications are computed in parallel. The orthogonal vectors produced in the QR decomposition

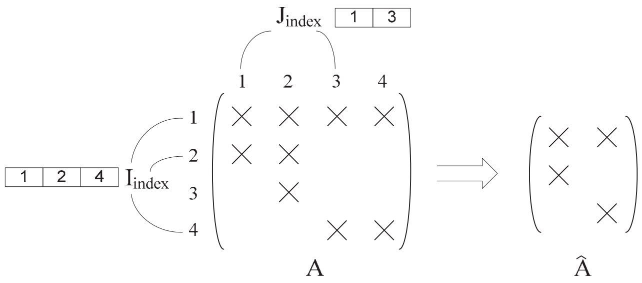


Fig. 3. Constructing local $\hat{A}$ matrices.

algorithm are stored in global memory (Q in Table 1) and used in proceeding steps. At the end of the *Compute-GSAI* stage, m_k values are computed via $\hat{m}_k = R^{-1}Q^T\hat{e}_k$ (R is the upper triangular matrix from the decomposition [1]) and scattered to global memory space allocated to M.

3.1.3 Post-GSAI Stage

Modify and assemble M. The values and row indices of the preconditioner generated in the *compute preconditioner* kernel are stored in the M_{value} and M_{index} vectors in the format shown in Fig. 4. Since the allocated size to each column of M on global memory is equal to $n_{2,max}$ (which is not necessarily equal to the number of nonzeros per column), to assemble M each warp has to store the number of nonzeros of the column it is generating into a vector called $M_{pointer}$. In the *Post-GSAI* stage the M_{value}, M_{index}, and $M_{pointer}$ data structures are modified to match the CSC storage format. The first kernel in the *Post-GSAI* stage called *modify* changes $M_{pointer}$ to match the CSC format (M^*_{pointer} in Fig. 4). Another kernel called *assemble* then modifies the M_{index} and M_{value} vectors on the GPU to match the column storage format (M^*_{index} and M^*_{value} in Fig. 4). The updated vectors are generated on GPU memory and do not need to be transferred to the CPU.

3.1.4 The Solver

Preconditioned BiCGStab solver. When generating a right preconditioner M (via minimizing (2)) matrices are stored and generated in column storage format to reduce memory access latencies [1]. On the other hand, to achieve the best performance and increase coalesced memory accesses on the GPU, the matrices in the sparse matrix vector multiplication kernel should be stored in row storage format [41]. Thus prior to solving $Ax = b$ the matrices are converted to CSR format (to generate a left preconditioner the *CSC to CSR* stage in Fig. 2 should be removed since all matrices are generated and stored in CSR format). After the conversion step the BiCGStab kernel is called to solve $Ax = b$ using the produced M.

The preconditioned BiCGStab iterative solver on GPU is dominated by the multiple sparse matrix multiplies [1]. The CPU is only used for scalar updates in the algorithm and major computing kernels are implemented on the GPU. Since sparse matrix vector multiplication is the most time consuming operation in iterative solvers [40] it has to be accelerated efficiently on the GPU. We used the SMVM implementation from [38], [41] which is one of the fastest implementations of this kernel on GPUs. Other operations in the BiCGStab iterative solver have also been accelerated on the GPU using CUBLAS [42] functions.

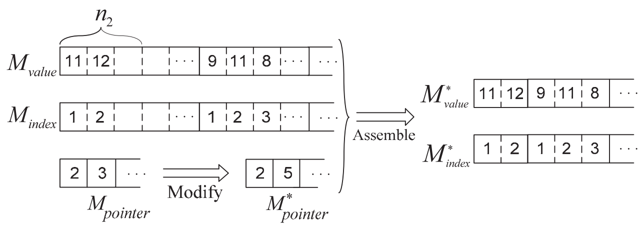


Fig. 4. $M_{pointer}$ is modified using the *Modify* kernel to match the CSC [2] storage format and then the *Assemble* kernel assembles the matrix values and stores them in CSC format (M^*_{index} and M^*_{value}).

3.2 Memory Allocation

In this section, we introduce techniques to overcome GPU memory space limitations and enable the correct implementation of the GSAI stages proposed in Section 3.1 for large problems. Since the exact size of data structures (such as $\hat{A}$ and Q) used in the *compute preconditioner* kernel are only determined during the kernel execution, techniques to allocate memory statically to these data structures in the *Pre-GSAI* stage (prior to calling the kernel) are also proposed. Based on the allocated memory space to each data structure, the number of *compute preconditioner* kernel calls required to generate the preconditioner are also determined. The implementations proposed in this section are all a part of the *allocate memory* section of the *Pre-GSAI* stage shown in Fig. 2.

Local data structures such as $\hat{A}$ and Q are generally large and cannot be stored on GPU shared memory; thus by approximating their size, global memory space is allocated to them in the *Pre-GSAI* stage prior to calling the *compute preconditioner* kernel. The maximum number of rows and columns in these matrices is computed in the *compute dimensions* kernel ($n_{1,max}$ and $n_{2,max}$) and global memory space equal to the size of an array with $n_{1,max} \times n_{2,max}$ elements is allocated to them per column (warp). The I_{index} vector used in the *Compute-GSAI* kernel also varies in size for each warp and can easily exceed the maximum size of shared memory. This vector is also stored in global memory by allocating memory to arrays of $n_{1,max}$ elements per column. To compute the preconditioner different columns of A are required thus the A matrix should be on global memory at all times. From Table 1, the amount of global memory required to store various vectors and data structures prior to calling the *Compute-GSAI* kernel are computed. For large A matrices and τ parameters that lead to a denser preconditioner, the total size of the data structures in Table 1 will exceed the GPU global memory. Since the memory required to store $\hat{A}$ and Q for all columns is considerably larger than the size of A and M, by calling multiple kernels sequentially and overwriting the memory space allocated to these matrices, computing the SAI preconditioner is made possible on the GPU. After storing A and M on the device depending on the available memory and size of other data structures that need to be on global memory, the computation of the preconditioner is divided between multiple kernels each producing a few columns of M. Thus, memory allocated to other data structures such as $\hat{A}$ and Q can be reused.

The small size of the GPU shared memory does not limit the size of the problem being solved, because large data structures in the kernel are stored on GPU global memory. To accelerate computations shared memory is used to store

TABLE 2
Properties of Tested Sparse Matrices

Matrix Name	Matrix Type	Rows	non-zeros
venkat01	CFD sequence	62,424	1,717,792
majorbasis	Optimization	160,000	1,750,416
t2em	Electromagnetic	921,632	4,590,832
atmosmodd	CFD	1,270,432	8,814,880
thermal2	thermal	1,228,045	8,580,313
g3_circuit	circuit simulation	1,585,478	7,660,826
apache2	structural	715,176	4,817,870

local data structures in the *compute preconditioner* kernel whenever possible. Before calling the *compute preconditioner* kernel, the amount of shared memory required for each block to store these data structures is checked and if it reduces the number of active blocks per SM to two all data is read from global memory directly. For larger tolerances which lead to larger data structures most of the data is read directly from global memory. Thus, the number of active blocks per SM is no longer limited to the size of data structures and shared memory and memory access latencies are reduced via configuring the L1 cache to 48 KB. To generate SAI preconditioners for very large problems which do not fit on the GPU global memory or to generate very dense preconditioners, the computation of the preconditioner should be distributed between many GPUs.

4 Results

The performance of the proposed GSAI technique is evaluated using seven matrices [43] from various application areas with different sparsity patterns (see Table 2). These problems are generally difficult to solve and precondition due to their complex geometry and ill-conditioning. GPU results were achieved using NVIDIA GTX480, TESLA M2070, and CUDA-SDK 3.2., CPU programs are executed on a system core Linux cluster from Sharcnet [44] using 1-32 AMD Opteron 252 (2.6 GHZ, single core) processors with a Quadrics Elan4 interconnect. The preconditioned BiCGStab iterations are terminated upon reaching 10,000 iterations or reaching a relative residual of less than 1e-7 in under 10,000 iterations using a random right-hand side (RHS) for all problems (the same RHS is used for each matrix in all platforms). Both the preconditioner generation kernel and the iterative solver run in double precision. In the following, the performance of the proposed GSAI preconditioner on GTX480 and TESLA M2070 is first presented (see Section 4.1), the preconditioner computation time on the GPU is then compared to *ParaSails* (see Section 4.2) on a single processor/core (a processor/core is an AMD Opteron 252 consisting of one core).

ParaSails computes the preconditioner in parallel on multiprocessor platforms by partitioning M and allocating the computation of its columns to different processors. They propose novel techniques to partition columns/rows among processors, hide interprocessor communication latencies, balance load among processors, manage one-sided communications, construct $\hat{A}$ matrices and perform operations such as QR decomposition. Implementation details of how the computation of SAI preconditioners is parallelized in *ParaSails* can be found in the documentations and publications

TABLE 3
The Effect of Increasing Tolerance (τ) on Iterations
(GSAI on GTX480)

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	65	59	50	45	70
majorbasis	49	47	49	43	23
t2em	2390	2390	2390	1264	1264
atmosmodd	268	268	268	145	145
thermal2	6000	5805	5727	3608	2906
g3_circuit	1856	2307	1863	1347	1145
apache2	2922	1674	1674	1143	1226
average	1936	1793	1717	1085	968

referenced in [7]. The time to compute the SAI preconditioner using GSAI on GPUs is compared to *ParaSails* on a cluster of multiple AMD Opteron 252 processors in Section 4.2.

4.1 The GSAI Preconditioning Method

In this section, the effect of increasing the tolerance τ in (3) using GSAI and NVIDIA GTX480 on the preconditioner construction time, iterative solver execution time and the number of iterations are first studied. The total execution time and the number of iterations of the preconditioned iterative solver are then presented for both GTX480 and TESLA M2070. As shown in Table 3 for larger tolerances (τ), the number of iterations considerably decreases for most of the tested problems using GSAI. Because the preconditioner M is an approximation of A^{-1} decreasing its sparsity using τ does not necessarily guarantee a better preconditioner, for example, the number of iterations in g3_circuit increases when τ is increased to 0.6 (see Table 3). But, on average, the number of iterations decrease as τ increases and the sparsity of M gets closer to A [4]. For most of the tested problems, the total execution time on GPU also decreases as τ increases (see Table 4). Because more elements of A satisfy the condition in (3) the maximum number of rows and columns ($n_{1,max}$ and $n_{2,max}$) of the local $\hat{A}$ matrices on the GPU increase with tolerance (see Fig. 5a). As a result the time required by the *compute dimensions* kernel to determine $n_{1,max}$ and $n_{2,max}$ as well as the time required to construct and decompose $\hat{A}$ in the *compute preconditioner* kernel also increase with τ (Fig. 5b and Table 5). Fig. 5b shows the fraction of total preconditioner execution time spent in all kernels involved in the construction of the SAI preconditioner on GTX480. Based on Table 5 for all tested matrices the preconditioner execution time increases with τ. Thus, except for copying A to the GPU, the execution time of all kernels increases with τ due to an increase in the number of non-zeros in preconditioner M (Fig. 5b and Table 5).

Fig. 6a and Table 4 explain why an increase in the SAI computing time for larger tolerances still on average improves the total execution time on GPU. As shown in Fig. 6a, the total execution time is dominated by the BiCGStab solver. Thus, based on total execution times reported in Table 4, by increasing τ and generally generating a more accurate preconditioner, the execution time of the iterative solver is decreased (due to an average reduced number of iterations) with a negligible increase in SAI computation time. Since the time spent in generating the

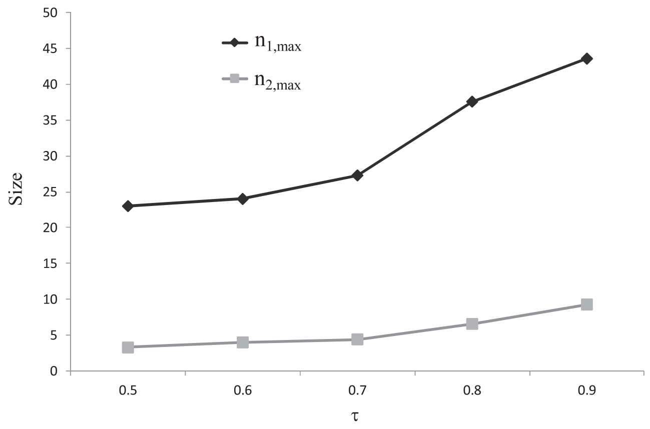
TABLE 4
The Effect of Increasing Tolerance (τ) on the Total Execution Time (Preconditioner Construction and Solve) (GSAI on GTX480)

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	0.43	0.54	0.69	0.83	2.6
majorbasis	0.66	0.64	0.65	0.68	0.6
t2em	108	108	108	59	59
atmosmodd	17	17	17	11	11
thermal2	364	348	331	213	174
g3_circuit	136	170	138	101	87
apache2	110	63	63	44	47
average	105	101	94	61	54

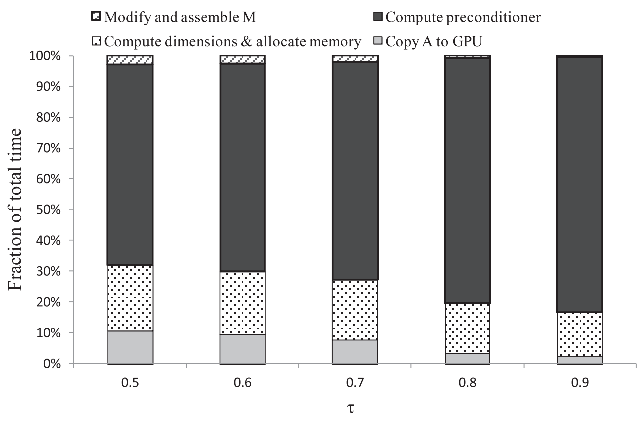
TABLE 5
The Effect of Increasing τ in the GSAI Algorithm (on GTX480) on the Preconditioner Construction Time

Matrix	$\tau = 0.5$	$\tau = 0.6$	$\tau = 0.7$	$\tau = 0.8$	$\tau = 0.9$
venkat01	0.11	0.24	0.42	0.58	2.14
majorbasis	0.11	0.11	0.11	0.19	0.3
t2em	0.3	0.3	0.3	1.26	1.26
atmosmodd	0.43	0.43	0.43	1.97	1.97
thermal2	0.42	0.42	0.7	1.65	2.7
g3_circuit	0.65	0.78	0.9	1.51	1.84
apache2	0.31	0.35	0.35	0.78	0.8
average	0.33	0.38	0.46	1.13	1.57

preconditioner is considerably less than the time required to solve the problem, the total execution time on average decreases for larger tolerance parameters. The problem solution time on the GPU decreases when the iterations are reduced on the GPU. This is because the SpMV kernel in the iterative solver uses available GPU resources more efficiently as the number of nonzeros in M increase. While the preconditioner becomes denser with larger τ parameters, the number of rows in M is fixed; as a result the number of computing blocks/warps launched on the GPU remain unchanged (because of using SpMV implementations proposed in [38] and [41]). On the other hand,

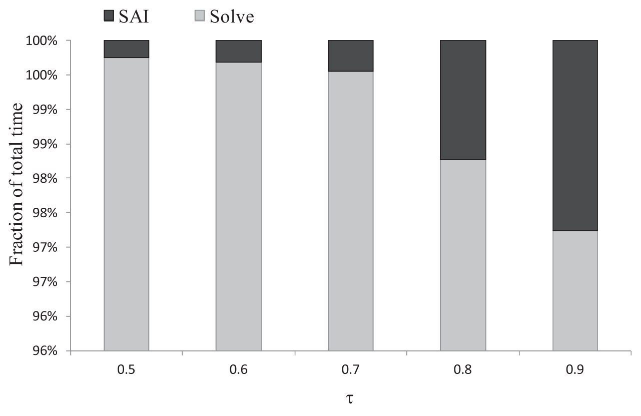


(a)

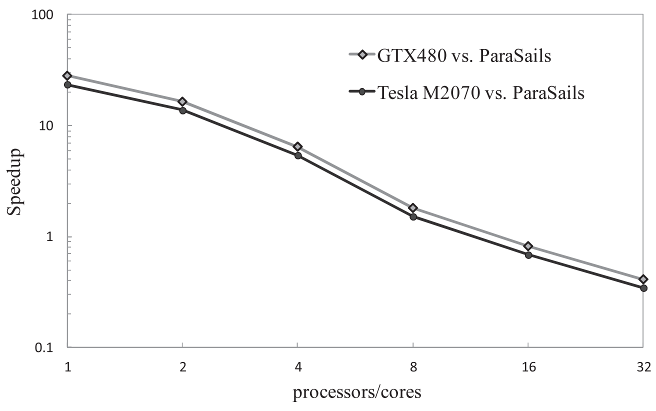


(b)

Fig. 5. The left figure (a) shows the effect of increasing τ on the maximum dimension of local $\hat{A}$ matrices ($n_{1,max}$, and $n_{2,max}$); in the right figure (b) the average fraction of total time (over all matrices) spent in the functions/kernels involved in the first three stages of the GSAI preconditioning algorithm (on GTX480) are shown for an increasing τ (*compute preconditioner* consists of all steps in the *Compute-GSAI* stage).



(a)



(b)

Fig. 6. The average fraction of total time (over all matrices) in generating the SAI preconditioner (the *Pre-GSAI*, *Compute-GSAI*, and *Post-GSAI* stages in Fig. 2) and solving the problem for an increasing τ on the GPU using GSAI (left figure, a); the right figure (b) shows the speedup achieved from generating the preconditioner on GTX480 and TESLA M2070 compared to generating the same preconditioner using *ParaSails* [7] on 1-32 processors/cores (the generated preconditioner has the same sparsity as A, $\tau = 1$ in GSAI).

TABLE 6
The Time Spent in Computing the Stages in Fig. 2 for $\tau = 0.9$ on GTX480

Matrix	Pre-GSAI $\tau = 0.9$	Compute-GSAI $\tau = 0.9$	Post-GSAI $\tau = 0.9$	Solve
venkat01	0.2	1.94	0.01	0.46
majorbasis	0.06	0.23	0.01	0.3
t2em	0.22	1.03	0.01	57
atmosmodd	0.389	1.57	0.02	9.5
thermal2	0.46	2.23	0.02	171
g3_circuit	0.33	1.49	0.02	85
apache2	0.17	0.62	0.01	47

the number of nonzeros per row increases, exploiting more parallelism per warp and better utilizing the GPU resources. In conclusion, GPU acceleration of SAI preconditioners allows for the generation of more accurate and denser preconditioners and increases the applicability of static SAI preconditioning.

Table 6 shows the execution time of the steps involved in constructing the SAI preconditioner on GTX480 for τ equal to 0.9 (which generated the best preconditioner among the tested tolerances) as well as the BiCGStab iterative solver. The time spent in constructing the preconditioner is less than 3 seconds for all matrices (see Table 6) while the iterative solver can take up to 171 seconds for matrices such as thermal2 on the GPU.

Preconditioners with more than 6 million nonzeros (see Table 7) are generated in less than 3 seconds (see Table 6) using the proposed GSAI technique. As shown in Table 7, without the preconditioner most of the problems would not converge in 10,000 iterations while the preconditioned BiCGStab solver would converge to the 1e-7 residual error in less than 100 iterations for some matrices (venkat01 and majorbasis). Table 7 also shows that although the number of iterations for the preconditioned solver on TESLA M2070 decreases compared to GTX480, the total execution time is still lager for the tested matrices. Architectural differences among the two GPUs (GTX480 versus TESLA M2070: 480 versus 448 cores, 1.4 GB versus 1.15-GB processor clock, 177 GB/s versus 150 GB/s bandwidth and floating point precision differences) are accountable for the iteration count and timing differences in the results.

4.2 GSAI versus ParaSails

In this section, the preconditioner construction time is compared with *ParaSails* [7] which also uses a priori techniques to determine the sparsity of M and computes SAI in parallel on multiprocessors. Techniques proposed in *ParaSails* to better determine the sparsity of M prior to its computations for PDE problems can be implemented in the *Pre-GSAI* stage of GSAI without changing the *compute preconditioner* kernel itself (determining the sparsity of M in a priori SAI preconditioning techniques is negligible compared to the preconditioner computation itself). To compare GSAI with *ParaSails*, parameters were set so that both *ParaSails* and GSAI would produce similar preconditioners with the same sparsity as A ($\tau = 1$ in GSAI, parameter settings for *ParaSails* are described in [7]), preconditioners are produced using unfactorized preconditioning in *ParaSails*. Table 8 shows generating the SAI preconditioner using *ParaSails* on one processor/core can take up to 100 seconds while the proposed acceleration of SAI preconditioners on GPUs generated the same preconditioner in less than 3 seconds. With GSAI on GTX480, speedups of up to 47 times are achieved compared to *ParaSails*, decreasing the average generation time of SAI preconditioners 28 times. In Fig. 6b, the average execution time of *ParaSails* for all matrices on multiprocessors is compared to average preconditioner generation time of GSAI on NVIDIA GTX480 and TESLA M2070. Fig. 6b shows constructing the preconditioner on a single GPU using GSAI is equivalent to constructing the same preconditioner on 16 processors/cores using *ParaSails*. GSAI computes many columns of M in parallel, the time spent to construct local $\hat{A}$ matrices do not accumulate for columns generated simultaneously. This is not the case in *ParaSails* when run on a single processor, so both the parallel execution of columns on the GPU and the techniques proposed to compute each column of M are the main reasons for the reported speedups.

TABLE 7
Preconditioned and Unpreconditioned BiCGStab Iterative Solver on GTX480 and TESLA M2070

Matrix	GPU BiCGStab Iterations	Precond. non-zeros	GTX480 Precond. BiCGStab Iterations	GTX480 Total Time	TESLA 2070 Precond. BiCGStab Iterations	TESLA M2070 Total Time
venkat01	>10000	822937	70	2.6	70	2.7
majorbasis	>10000	646524	23	0.6	23	0.72
t2em	>10000	4590832	1264	59	968	63
atmosmodd	>10000	6317824	145	11	140	14
thermal2	6119	6720218	2906	174	2804	195
g3_circuit	>10000	6562707	1145	87	1133	108
apache2	4931	2677127	1226	47	1115	58

The table shows the number of iterations (column one) required to solve the unpreconditioned BiCGStab solver for the tested matrices, the number of nonzeros in the preconditioner produced for $\tau = 0.9$ and the iterations achieved from the preconditioned BiCGStab solver using this preconditioner on both the GTX480 and TESLA M2070 graphic cards.

TABLE 8
ParaSails Execution Time Compared to GPU Results

Matrix	*ParaSails*-Setup	*ParaSails*-Preconditioner	*ParaSails*-Total	GTX480 $\tau = 1$	*ParaSails*-Total vs. GTX480 speedup	TESLA M2070 $\tau = 1$	*ParaSails*-Total vs. TESLA M2070 speedup
venkat01	0.1	13.7	13.8	2.22	6.2	2.83	4.8
majorbasis	0.1	14.7	14.8	1.19	12.3	1.43	10.3
t2em	0.4	60	60.4	1.26	47.9	1.55	38.9
atmosmodd	0.7	93.7	94.4	3	31	3.8	24.8
thermal2	0.8	91.7	92.5	3.76	24.5	3.9	23.7
g3_circuit	0.7	99.4	100.1	2.13	46.8	2.64	37
apache2	0.4	52	52.4	1.62	32.3	2	25.8
average speedup	--	--	--	--	28.7	--	23.7

The time to setup (ParaSails-Setup) and compute (ParaSails-Preconditioner) the SAI preconditioner with the same sparsity as A ($\tau = 1$ in GSAI) on ParaSails for one processor/core compared to the time required to compute the preconditioner on GTX480 (GPU-SAI) and TESLA M2070 using the GSAI preconditioning algorithm (ParaSails-Total is computed by adding ParaSails-Setup and ParaSails-Preconditioner).

5 Conclusion and Future Work

The proposed GPU accelerated SAI preconditioning method (GSAI) introduces optimized implementations to parallelize the computation of SAI preconditioners on NVIDIA GPUs. The effects of decreasing the sparsity of the preconditioner using a tolerance parameter τ are also tested on the GPU using GSAI. The results showed that the number of iterations and total execution time would on average decrease using GSAI for larger tolerances; the preconditioner generation time would remain negligible compared to the problem solution time. The total execution time on the GPU (the time spent on generating the preconditioner and solving the problem) would constantly decrease as τ increases making the generation of denser preconditioner more efficient. The generation of the SAI preconditioner was accelerated on average 28 and 23 times faster on GTX480 and TESLA M2070, respectively, using GSAI compared to the time required to create the same preconditioner using *ParaSails* on a single processor (single-core AMD Opteron 252). The preconditioner generation time on GTX480 and TESLA M2070 (using GSAI) is almost equivalent to creating the SAI preconditioner on 16 processors in parallel using *ParaSails*. We plan to accelerate the execution time of other variants of SAI preconditioning techniques such as adaptive methods and also introduce techniques to find better approximations of the preconditioner using GPUs in future work.

Acknowledgments

This work was partly supported by the Natural Sciences Engineering Research Council of Canada and the National Science Foundation under grant no. CCF-1065448. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Foundations.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems,* pp. 10-349, SIAM, 2003.

[2] J.M. Jin, *The Finite Element Method in Electromagnetics,* pp. 19-44, Wiley-IEEE Press, 2002.

[3] R. Barrett et al., *Templates for the Solution of Linear Systems.* SIAM, 1994.

[4] J. Zhongxiao and Z. Baochen, "A Power Sparse Approximate Inverse Preconditioning Procedure for Large Sparse Linear Systems," *Numerical Linear Algebra with Applications,* vol. 16, no. 4, pp. 259-299, 2009.

[5] E. Chow, "Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners with a Priori Sparsity Patterns," *Int'l J. High Performance Computing Applications,* vol. 15, no. 1, pp. 56-74, 2001.

[6] E. Chow, "A Priori Sparsity Patterns for Parallel Sparse Approximate Inverse Preconditioners," *SIAM J. Scientific Computing,* vol. 21, no. 5, pp. 1804-1822, 1999.

[7] https://computation.llnl.gov/casc/parasails/parasails.html, 2013.

[8] T. Huckle et al., "An Efficient Parallel Implementation of the MSPAI Preconditioner," *Parallel Computing,* vol. 36, nos. 5/6, pp. 273-284, 2010.

[9] M.J. Grote and T. Huckle, "Parallel Preconditioning with Sparse Approximate Inverses," *SIAM J. Scientific Computing,* vol. 18, no. 3, pp. 838-853, 1997.

[10] P. Raghavan and K. Teranishi, "Parallel Hybrid Preconditioning: Incomplete Factorization with Selective Sparse Approximate Inversion," *SIAM J. Scientific Computing,* vol. 32, no. 3, pp. 1323-1345, 2010.

[11] P. Gonzaléz, T.F. Pena, and J.C. Cabaleiro, "Parallel Sparse Approximate Preconditioners Applied to the Solution of BEM Systems," *Eng. Analysis with Boundary Elements,* vol. 28, no. 9, pp. 1061-1068, 2004.

[12] M. Benson et al., "Parallel Algorithms for the Solution of Certain Large Sparse Linear Systems," *Int'l J. Computer Math.,* vol. 16, nos. 3/4, pp. 245-260, 1984.

[13] S.T. Barnard, L.M. Bernardo, and H.D. Simon, "An MPI Implementation of the SPAI Preconditioner on the T3E," *Int'l J. High Performance Computing Applications,* vol. 13, no. 2, pp. 107-123, 2010.

[14] G.A. Gravavis et al., "Finite Element Approximate Inverse Preconditioning Using POSIX Threads on Multicore Systems," *Proc. Int'l Multiconf. Computer Science and Information Technology,* pp. 297-302, 2010.

[15] G.A. Gravavis, "High Performance Inverse Preconditioning," *Archives of Computational Methods in Eng.,* vol. 16, no. 1, pp. 77-108, 2009.

[16] K. Xu et al., "FSAI Preconditioned CG Algorithm Combined with GPU Technique for the Finite Element Analysis of Electromagnetic Scattering Problems," *Finite Elements in Analysis and Design,* vol. 47, no. 4, pp. 387-393, 2011.

[17] J.D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, 2007.

[18] http://developer.nvidia.com/cuda-toolkit-32-downloads, 2013.

[19] D.F. Cosgrove, J.C. Dias, and A. Griewank, "Approximate Inverse Preconditioning for Sparse Linear Systems," *Int'l J. Computer Math.,* vol. 44, no. 1-4, pp. 91-110, 1992.

[20] E. Chow and Y. Saad, "Approximate Inverse Preconditioners via Sparse-Sparse Iterations," *SIAM J. Scientific Computing,* vol. 19, no. 3, pp. 995-1023, 1998.

[21] http://www.computational.unibas.ch/software/spai/spaidoc.html, 2013.

[22] M.J. Grote and H.D. Simon, "Parallel Preconditioning and Approximate Inverses on the Connection Machine," *Parallel Processing for Scientific Computing,* vol. 2, pp. 519-523, 1992.

[23] T. Huckle, "Approximate Sparsity Patterns for the Inverse of a Matrix and Preconditioning," *Proc. IMACS World Congress on Scientific Computation,* 1997.

[24] L. Kolotilina, "Explicit Preconditioning of Systems of Linear Algebraic Equations with Dense Matrices," *SIAM J. Matrix Analysis and Applications,* vol. 13, pp. 2566-2573, 1992.

[25] L. Kolotilina and A.Y. Yeremin, "Factorized Sparse Approximate Inverse Preconditionings I: Theory," *SIAM J. Matrix Analysis and Applications,* vol. 14, no. 1, pp. 45-58, 1993.

[26] M. Benzi, C.D. Meyer, and M. Tuma, "A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method," *SIAM J. Scientific Computing,* vol. 17, no. 5, pp. 1135-1149, 1998.

[27] M. Benzi and M. Tuma, "A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems," *SIAM J. Scientific Computing,* vol. 19, no. 3, pp. 968-994, 1998.

[28] M. Benzi and M. Tuma, "A Comparative Study of Sparse Approximate Inverse Preconditioners," *Applied Numerical Math.,* vol. 30, no. 2/3, pp. 305-340, 1999.

[29] M. Bollhöfer and V. Mehrmann, "Algebraic Multilevel Methods and Sparse Approximate Inverses," *SIAM J. Matrix Analysis and Applications,* vol. 24, no. 1, pp. 191-218, 2002.

[30] M. Bollhöfer and Y. Saad, "A Factored Approximate Inverse Preconditioner with Pivoting," *SIAM J. Matrix Analysis and Applications,* vol. 23, no. 3, pp. 692-705, 2001.

[31] S. Kharchenko et al., "A Robust AINV-Type Method for Constructing Sparse Approximate Inverse Preconditioners in Factored Form," *Numerical Linear Algebra with Applications,* vol. 8, no. 3, pp. 165-179, 2001.

[32] M. Benzi and M. Tuma, "A Comparative Study of Sparse Approximate Inverse Preconditioners," *J. Applied Numerical Math.,* vol. 30, no. 2/3, pp. 305-340, 1999.

[33] W. Tang, "Towards an Effective Sparse Approximate Inverse Preconditioner," *SIAM J. Matrix Analysis and Applications,* vol. 20, no. 4, pp. 970-986, 1999.

[34] J. Cosgrove et al., "Structural Properties of the Graph of Augmented Sparse Approximate Inverses," *Proc. Symp. Applied Computing,* pp. 131-136, 1990.

[35] G. All'eon et al., "Sparse Approximate Inverse Preconditioning for Dense Linear Systems Arising in Computational Electromagnetics," *Numerical Algorithms,* vol. 16, no. 1, pp. 1-15, 1997.

[36] W. Tang and W. Wan, "Sparse Approximate Inverse Smoother for Multigrid," *SIAM J. Matrix Analysis and Applications,* vol. 21, no. 4, pp. 1236-1252, 2000.

[37] M. Mehri Dehnavi, D. Fernandez, and D. Giannacopoulos, "Finite Element Sparse Matrix Vector Multiplication on GPUs," *IEEE Trans. Magnetics,* vol. 46, no. 8, pp. 2982-2985, Aug. 2010.

[38] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," technical report, NVIDIA, 2008.

[39] T.D. Davis, *Direct Methods for Sparse Linear Systems,* pp. 7-17, SIAM, 2006.

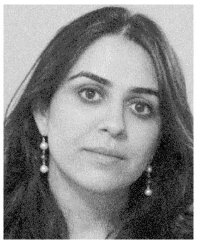
[40] M. Mehri Dehnavi, D. Fernandez, and D. Giannacopoulos, "Enhancing the Performance of Conjugate Gradient Solvers on Graphic Processing Units," *IEEE Trans. Magnetics,* vol. 47, no. 5, pp. 1162-1165, May 2011.

[41] http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUSPARSE_Library.pdf, 2013.

[42] http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf, 2013.

[43] http://www.cise.ufl.edu/research/sparse/matrices, 2013.

[44] https://www.sharcnet.ca, 2013.



Maryam Mehri Dehnavi received the BSc degree in electrical engineering from Isfahan University of Technology, Iran, in 2005 and the MSc degree in computer engineering from the University of Calgary, Canada, in 2007. She is currently working toward the PhD degree at the Department of Electrical and Computer Engineering at McGill University, Montreal, Canada and a visiting researcher at the Parallel Computing Laboratory (ParLab) in the University of California Berkeley. From January 2011 to April 2011, she was a visiting scholar at the Parallel Systems and Computer Architecture Lab (PASCAL), University of California, Irvine. She was the recipient of the 2009 Alexander Graham Bell Canada Graduate Scholarship (CGS3 Doctoral) from the Natural Science and Engineering Council of Canada (NSERC), the 2010 international internship scholarship of the Fonds Nature et Technologies, McGill 2011 visiting researcher travel grant, Best Student Paper Award in IEEE Conference on Computational Electromagnetics (Compumag 2009), 2008 McGill graduate scholarship and 2005 University of Calgary graduate scholarship. Her research interests include: scientific computing, general-purpose computation on graphics hardware, computer architecture/engineering, parallel computing, numerical analysis, and computer aided design for electromagnetic simulations. She is a member of the IEEE.

David M. Fernández received the BSc degree in electrical engineering in la Universidad del Zulia, LUZ, Maracaibo-Venezuela, 1998, the MSc degree in applied computing, LUZ, in 2002, and the PhD degree in electrical and computing engineering from McGill University, Montreal, Canada, in 2011. He currently works as a professor at LUZ, where he teaches microprocessors architecture and numerical methods for electrical engineering. He has served as a reviewer for *IEEE Transactions of Magnetics*. He graduated as the best student of 1998, also obtained the best poster award at CCPW-2009 (Carlton University) and later the best student talk at HPCS-2011 (Montreal-Canada). His main research topics include hardware acceleration of electromagnetic computations through different multi/many core architectures, sparse matrix computations, and PLC emulation. She is a member of the IEEE.

Jean-Luc Gaudiot received the diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France, in 1976 and the MS and PhD degrees in computer science from the University of California, Los Angeles, in 1977 and 1982, respectively. He is currently a professor and chair of the Electrical and Computer Engineering Department at the University of California, Irvine. Prior to joining UCI in January 2002, he was a professor of electrical engineering at the University of Southern California since 1982, where he served as and director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979-1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980-1982). He consults for a number of companies involved in the design of high-performance computer architectures. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published more than 200 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial organizations. In January 2006, he became the first editor-in-chief of IEEE Computer Architecture Letters, a new publication of the IEEE Computer Society, which he helped found to the end of facilitating short, fast turnaround of fundamental ideas in the Computer Architecture domain. From 1999 to 2002, he was the editor-in-chief of the *IEEE Transactions on Computers*. In June 2001, he was elected chair of the IEEE Technical Committee on Computer Architecture, and reelected in June 2003 for a second 2-year term. He has also chaired the IFIP Working Group 10.3 (Concurrent Systems). He is one of three founders of PACT, the ACM/IEEE/IFIP Conference on Parallel Architectures and Compilation Techniques, and served as its first program chair in 1993, and again in 1995. He has also served as a program chair of the 1993 Symposium on Parallel and Distributed Processing, HPCA-5 (1999 High Performance Computer Architecture), the 16th Symposium on Computer Architecture and High Performance Computing (Foz do Iguacu, Brazil), the 2004 ACM International Conference on Computing Frontiers, and the 2005 International Parallel and Distributed Processing Symposium. He is a member of the IEEE, the ACM, and the ACM SIGARCH. In 1999, he became a fellow of the IEEE. He was elevated to the rank of AAAS fellow in 2007.

Dennis D. Giannacopoulos received the BEng and PhD degrees in electrical engineering from McGill University, Montreal, Canada, in 1992 and 1999, respectively. He has been with the Department of Electrical and Computer Engineering at McGill University since 2000, where he is currently an associate professor and a member of the Computtional Electromagnetics Group. He has been the recipient of his department's professor of the Year Award twice. His research interests include parallel adaptive finite element analysis for electromagnetics and the acceleration of computational electromagnetics algorithms on emerging parallel architectues. He has authored or coauthored more than 85 referred journal and conference publications. His students have received three best paper/presentation awards at international conferences and symposia. His research has been sponsored by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds de recherche du Québec-Nature et technologies (FQRNT), and the Canada Foundation for Innovation (CFI). He has served on the editorial boards and technical program committees of several major international conferences and served as cochair of the editoral board for the 14th Conference on the Computation of Electromagnetic Fields. He is a member of the IEEE, the International Compumag Society, and the Ordre des Ingéniuers du Québec.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.