INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING, *SERIES B*
Volume 5, Number 1-2, Pages 13–20

DEVELOPMENT OF A RESTRICTED ADDITIVE SCHWARZ PRECONDITIONER FOR SPARSE LINEAR SYSTEMS ON NVIDIA GPU

HUI LIU, ZHANGXIN CHEN, SONG YU, BEN HSIEH AND LEI SHAO

Abstract. In this paper, we develop, study and implement a restricted additive Schwarz (RAS) preconditioner for speedup of the solution of sparse linear systems on NVIDIA Tesla GPU. A novel algorithm for constructing this preconditioner is proposed. This algorithm involves two phases. In the first phase, the construction of the RAS preconditioner is transformed to an incomplete-LU problem. In the second phase, a parallel triangular solver is developed and the incomplete-LU problem is solved by this solver. Numerical experiments show that the speedup of this preconditioner is sufficiently high.

Key words. Restricted additive Schwarz preconditioner, linear solver, ILU, parallel triangular solver, GPU

1. Introduction

A restricted additive Schwarz (RAS) preconditioner is a general parallel preconditioner for speedup of the solution of sparse linear systems, which was developed by Cai et al. [4]. This preconditioner is a cheaper variant of the classical additive Schwarz preconditioner, and it is faster in terms of iteration counts and CPU time [4, 5]. Nowadays, RAS is the default parallel preconditioner for the solution of nonsymmetric sparse linear systems in PETSc [1, 4] and has been used in PHG [20]. Our long-term goal is to develop and implement this type of preconditioners for numerical reservoir simulation [7, 8].

GPU, which was used only for graphics processing in its earlier development, is now much more powerful in float point calculation than conventional CPU. It has been used in many scientific applications, such as FFT [16], BLAS [2, 3, 16], Krylov subspace solvers [18, 13, 14, 15] and algebraic multigrid solvers [11]. Algorithms for basic matrix and vector operations are well understood now. However, due to the irregularity of sparse linear systems, the development of efficient parallel preconditioners on GPU is still challenging. In this paper, we introduce, study and implement a RAS preconditioner for speedup of the solution of sparse linear systems on NVIDIA Tesla GPU. For a given matrix A whose size is $n \times n$, a sub-problem is constructed and written as a smaller $i \times i$ matrix, $i \leq n$ [4]. Following this idea, combining all sub-problems together, the final problem becomes a diagonal block matrix problem, $\mathrm{diag}(A_1, A_2, \ldots, A_k)$, which can be solved by incomplete-LU factorization, where k is the number of sub-problems. We have recently developed a parallel triangular solver in [15], where a new matrix format, HEC (hybrid ELL and CSR), and a modified level schedule method on GPU have been introduced. This parallel triangular solver will be used in the current development and study of the RAS preconditioner. Numerical experiments performed show that the speedup of this preconditioner is sufficiently high.

Received by the editors January 5, 2014 and, in revised form, March 19, 2014.
1991 *Mathematics Subject Classification.* 65F08, 65F50, 65Y04, 65Y05, 65Y10.
This research was supported by NSERC/AIEE/Foundation CMG and AITF Chairs.

The layout is as follows: In §2, basic knowledge and our deduction of RAS are introduced. In §3, our parallel triangular solver is described. In §4, numerical experiments are employed to test our GPU version RAS preconditioner. In the end, some conclusions are presented.

2. Restricted Additive Schwarz Preconditioner

We consider a linear system:

$$Ax = b, \tag{1}$$

where $A = \big(A_{ij}\big)$ is an $n \times n$ nonsingular sparse matrix. Denote by L the lower part of A. Then the non-zero pattern we use is $L + L^T$, where L^T is the transpose of L. Also, we define an undirected graph $G = \{W, E\}$, where the set of vertices $W = \{1, \ldots, n\}$ represents the n unknowns and the edge set $E = \{(i, j) : A_{ij} \neq 0,\ A_{ij} \in L + L^T\}$ represents the pairs of vertices [4].

The graph G is partitioned into k non-overlapping subsets by METIS [12], denoted by $W_1^0, W_2^0, \ldots, W_k^0$. We always assume that all subsets of W are sorted in ascending order according to the column indices. For any subset W_i^0, a 1-overlap subset W_i^1 can be obtained by including all the immediate neighboring vertices in W [4]. Repeating this process, a δ-overlap subset W_i^δ can be defined, and the resulting overlapping subsets are $W_1^\delta, W_2^\delta, \ldots, W_k^\delta$.

For any nonempty subset V of W with N ($N > 0$) vertices, we define a mapping $m : V \to W$ by

$$m(p(j)) = j, \tag{2}$$

where $p(j)$ is the position of vertex j in V. Then we introduce matrix B as follows:

$$B_{ij} = A_{m(p(i))m(p(j))}. \tag{3}$$

In this case, B is an $N \times N$ matrix.

Applying this definition, for any W_i^δ with N_i vertices, we introduce the mappings $m_1, m_2, \ldots, m_k$. Using equation (3), we obtain submatrices, $A_1, A_2, \ldots, A_k$. When a RAS preconditioner is applied to the solution of linear systems, these submatrices can be solved simultaneously. We now assemble these submatrices and solve an enlarged system:

$$M = \operatorname{diag}(A_1, A_2, \ldots, A_k), \tag{4}$$

where M is an $(N_1 + N_2 + \ldots + N_k) \times (N_1 + N_2 + \ldots + N_k)$ matrix. This matrix can be solved by ILU(k) or ILUT. The structures of L and U are of the following form:

$$L = \operatorname{diag}(L_1, L_2, \ldots, L_k), \quad U = \operatorname{diag}(U_1, U_2, \ldots, U_k). \tag{5}$$

The final problem is how to solve the lower and upper triangular problems. The whole assembling procedure is described in Algorithm 1.

Algorithm 1 Assembling a RAS preconditioner

```
1: Constructing the undirected graph $G$ using pattern $L + L^T$;
2: Partitioning $G$ using METIS;
3: Constructing subgraph $W_i^\delta$ and the corresponding mapping $m_i$;
4: Assembling submatrix $A_i$;
5: Factorizing $A_i$ and obtaining the lower and upper triangular matrices $L_i$ and
   $U_i$, respectively.
```

3. Parallel Triangular Solver

For the sake of completeness, we will describe our triangular solver in this section. Problem (1) is factorized into the lower triangular problem

$$Ly = b \tag{6}$$

and the upper triangular problem

$$Ux = y; \tag{7}$$

i.e.,

$$LUx = b \Leftrightarrow Ly = b, \quad Ux = y, \tag{8}$$

where L and U are the lower and upper triangular matrices, respectively, b is the right-hand side and x is the unknown to be solved for. Since these two problems are similar to each other, we will only focus on the lower triangular problem (6). We always assume that each row of L is sorted in ascending order according to their column indices; in this case, the last element of each row is always the diagonal element.

3.1. Matrix Format. The matrix format we design is HEC (hybrid ELL and CSR). Its basic structure is demonstrated in Figure 1. An HEC matrix contains two submatrices: an ELL matrix, which was introduced in ELLPACK [10], and a CSR matrix (Compressed Sparse Row). The ELL matrix is stored in column-major order and is aligned when being stored in GPU. The CSR matrix is restricted in that each row has at least one element, which is a diagonal element for the triangular matrix L.

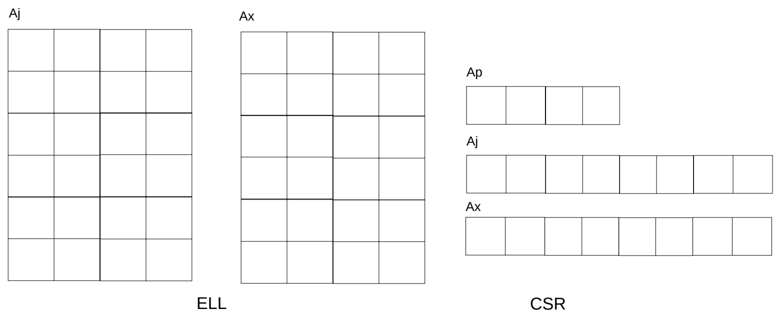

FIGURE 1. HEC matrix format.

When we split L, the row length of the ELL matrix is obtained by solving a minimum problem [3]:

$$\text{Find } l \ (l \geq 0) \text{ such that } w(i) = i * n + p_r * nz(i) \text{ is minimized,} \tag{9}$$

where p_r is the relative performance of the ELL and CSR matrices and $nz(i)$ is the number of non-zeros in the CSR part when the row length of the ELL part is i. A typical value for p_r is 20 [3].

3.2. Modified Level Schedule Algorithm. The parallel triangular solver algorithm we develop is based on the level schedule method [19, 14]. The idea is to group unknowns $x(i)$ into different levels so that all unknowns within the same level can be computed simultaneously [19, 14]. For the lower triangular problem, the level of $x(i)$ is defined as

$$l(i) = 1 + \max_j l(j) \text{ for all } j \text{ such that } L_{ij} \neq 0, \tag{10}$$

where L_{ij} is the (i, j)th entry of L and $l(i)$ is zero initially.

Define $S_i = \{x(j) : l(j) = i\}$, which is the union of all unknowns whose level is i. Here we also assume that set S_i is sorted in ascending order according to the indices of the unknowns that belong to S_i. Define by N_i the number of unknowns in set S_i and $nlev$ the number of levels. Now, a mapping $m(i)$ can be defined as follows:

$$m(i) = \sum_{j=1}^{k-1} N_j + p_k(x(i)), \quad x(i) \in S_k, \tag{11}$$

where $p_k(x(i))$ is the position $x(i)$ in the set S_k when $x(i)$ belongs to S_k.

Now, we reorder the triangular matrix L to L', where L_{ij} in L is transformed to $L'_{m(i)m(j)}$ in L'. L' is still a lower triangular matrix. From the mapping $m(i)$, we see that if $x(i)$ is next to $x(j)$ in the set S_k, then the ith and jth rows of L are next to each other in L' after reordering. It means that L is reordered level by level, which implies that memory access in matrix L' is less irregular than that in matrix L.

The parallel triangular solver algorithm is described in two steps, a preprocessing step and a solution step, respectively. The preprocessing step is described in Algorithm 2. In this step, the level of each unknown is calculated first. According to these levels, a mapping between L and L' can be set up according to equation (11). Then the matrix L is reordered and converted to the HEC format.

Algorithm 2 Preprocessing a lower triangular problem

```
1: Calculating the level of each unknown $x(i)$ using equation (10);
2: Calculating the mapping $m(i)$ using equation (11);
3: Reordering matrix $L$ to $L'$ using mapping $m(i)$;
4: Converting $L'$ to the HEC format.
```

The second step is to solve the lower triangular problem. This step is described in Algorithm 3, where $level(i)$ is the start row position of level i. First, the right-hand side vector b is permutated according to the mapping $m(i)$ we have just computed. Then the triangular problem is solved level by level and the solution in the same level is simultaneous. Each thread is responsible for one row. In the end, the final solution is obtained by another permutation.

4. Numerical Results

In this section, four examples will be tested, which are performed on our workstation with Intel Xeon X5570 CPU and NVIDIA Tesla C2050/C2070 GPUs. The operating system is Fedora 13 X86_64 with CUDA Toolkit 3.2 and GCC 4.4. All CPU codes are compiled with -O3 option. The type of float point number is double. The linear solver is GMRES(20).

Algorithm 3 Parallel lower triangular solver on GPU, $Lx = b$

```
1: for i = 1: n do                       ▷ Use one GPU kernel to deal with this loop
2:     $b'(m(i)) = b(i)$;
3: end for
4: for i = 1 : nlev do                                          ▷ Solve $L'x' = b'$
5:     start = level(i);
6:     end = level(i + 1) - 1;
7:     for j = start: end do             ▷ Use one GPU kernel to deal with this loop
8:         solve the $j$th row;
9:     end for
10: end for
11: for i = 1: n do                      ▷ Use one GPU kernel to deal with this loop
12:     $x(i) = x'(m(i))$;
13: end for
```

Example 1. The matrix used in this example is from a three-dimensional Poisson equation. The dimension is 1,728,000 and the number of non-zeros is 12,009,600. Performance data is collected in Table 1.

Table 1. Performance of the matrix from Poisson equation (C-t=CPU time (s), G-t=GPU time (s))

Preconditioner	Blocks	Overlap	C-t	G-t	Speedup	Iterations
RAS + ILU(0)	16	1	34.99	4.24	8.19	11
RAS + ILU(0)	512	1	41.35	4.77	8.65	12
RAS + ILU(0)	2048	1	47.80	5.01	9.49	12
RAS + ILU(0)	16	2	35.44	4.38	8.03	11
RAS + ILU(0)	512	2	45.18	5.00	9.00	11
RAS + ILU(0)	2048	2	54.06	5.65	9.52	11
RAS + ILUT	16	1	26.85	3.18	8.39	6
RAS + ILUT	512	1	26.51	3.69	7.13	7
RAS + ILUT	2048	1	39.50	4.48	8.77	8
RAS + ILUT	16	2	19.05	2.80	6.75	5
RAS + ILUT	512	2	26.33	3.72	7.02	6
RAS + ILUT	2048	2	30.95	4.37	7.06	6

From Table 1 we can see that when ILU(0) is applied as a solver, the average speedup of RAS is around 9. The number of iterations does not increase much when the number of blocks increases. This means that RAS is not very sensitive to the number of blocks in this example and we can obtain higher speedup by increasing the number of blocks.

When ILUT is applied, the total running time and the number of iterations are reduced compared to ILU(0), which coincides with theory [19]. ILUT is a better choice than ILU(0) usually. However, the non-zero pattern of L and U from ILUT is less regular than that of L and U from ILU(0) generally, and thus the speedup of ILUT may not be as high as that of ILU(0). For this problem, the average speedup of ILUT is over 7.

Example 2. Matrix atmosmodd is taken from the University of Florida sparse matrix collection [9] and is derived from a computational fluid dynamics problem.

The dimension of atmosmodd is 1,270,432 and it has 8,814,880 non-zeros. Performance data is collected in Table 2.

Table 2. Performance of atmosmodd (C-t=CPU time (s), G-t=GPU time (s))

Preconditioner	Blocks	Overlap	C-t	G-t	Speedup	Iterations
RAS + ILU(0)	16	1	20.39	2.50	7.95	8
RAS + ILU(0)	512	1	25.32	2.72	9.24	8
RAS + ILU(0)	2048	1	27.21	3.28	8.25	9
RAS + ILU(0)	16	2	21.02	2.69	7.77	8
RAS + ILU(0)	512	2	22.10	2.91	7.54	7
RAS + ILU(0)	2048	2	30.96	3.85	7.99	8
RAS + ILUT	16	1	19.80	2.46	8.00	6
RAS + ILUT	512	1	15.16	2.23	6.72	5
RAS + ILUT	2048	1	17.65	2.41	7.27	5
RAS + ILUT	16	2	16.59	2.28	7.23	5
RAS + ILUT	512	2	19.11	2.71	7.01	5
RAS + ILUT	2048	2	21.33	3.24	6.54	5

The data in Table 2 is similar to the data in Table 1. When ILU(0) is applied, the average speedup of RAS is around 8 while the average speedup of RAS is around 7 when ILUT is used as the solver. ILUT is better than ILU(0) in terms of the total running time and the number of iterations.

Example 3. Matrix atmosmodl is taken from the University of Florida sparse matrix collection [9] and is derived from a computational fluid dynamics problem. The dimension of atmosmodl is 1,489,752 and it has 10,319,760 non-zeros. Performance data is collected in Table 3.

Table 3. Performance of atmosmodl (C-t=CPU time (s), G-t=GPU time (s))

Preconditioner	Blocks	Overlap	C-t	G-t	Speedup	Iterations
RAS + ILU(0)	16	1	12.85	1.54	8.23	4
RAS + ILU(0)	512	1	15.88	1.60	9.81	4
RAS + ILU(0)	2048	1	15.19	1.70	8.82	4
RAS + ILU(0)	16	2	12.89	1.59	8.02	4
RAS + ILU(0)	512	2	14.95	1.83	8.09	4
RAS + ILU(0)	2048	2	16.84	2.08	8.00	4
RAS + ILUT	16	1	9.03	1.20	7.41	2
RAS + ILUT	512	1	14.08	1.56	8.87	3
RAS + ILUT	2048	1	15.44	1.71	8.90	3
RAS + ILUT	16	2	8.35	1.22	6.71	2
RAS + ILUT	512	2	9.94	1.35	7.23	2
RAS + ILUT	2048	2	13.62	1.61	8.36	2

For atmosmodl, when ILU(0) is applied and the number of blocks is set to 512, we have a maximal speedup of 9.8. When ILUT is applied, we have a maximal speedup of 8.9. The data shows that RAS is very stable in terms of the number of blocks. The ILUT solver is still slightly better than ILU(0). The average speedups for ILU(0) and ILUT are around 8.3 and 8, respectively.

Example 4. A matrix from SPE10 is applied. SPE10 is a standard benchmark for the black oil simulator [6]. The problem is highly heterogenous and it has been designed to be difficult to solve. The grid size for SPE10 is 60x220x85. The number of unknowns is 2,188,851 and the number of non-zeros is 29,915,573. Performance data is collected in Table 4.

Table 4. Performance of SPE10 (C-t=CPU time (s), G-t=GPU time (s))

Preconditioner	Blocks	Overlap	C-t	G-t	Speedup	Iterations
RAS + ILU(0)	8	0	102.17	15.75	6.48	23
RAS + ILU(0)	16	0	184.28	19.52	9.42	29
RAS + ILU(0)	512	0	123.02	17.81	6.90	28
RAS + ILU(0)	2048	0	188.04	27.28	6.89	44
RAS + ILU(0)	8	1	102.14	15.66	6.51	21
RAS + ILU(0)	16	1	123.20	16.65	7.39	22
RAS + ILU(0)	512	1	145.36	20.06	7.23	22
RAS + ILU(0)	2048	1	151.01	24.74	6.09	23
RAS + ILU(0)	8	2	106.02	17.07	6.20	21
RAS + ILU(0)	16	2	115.95	18.18	6.37	21
RAS + ILU(0)	512	2	146.16	24.87	5.87	21
RAS + ILU(0)	2048	2	220.73	34.04	6.48	23
RAS + ILUT	8	0	37.20	10.72	3.46	7
RAS + ILUT	16	0	51.83	13.49	3.84	10
RAS + ILUT	512	0	76.97	14.98	5.13	16
RAS + ILUT	2048	0	98.59	17.36	5.67	21
RAS + ILUT	8	1	28.84	8.42	3.41	5
RAS + ILUT	16	1	36.97	9.38	3.93	6
RAS + ILUT	512	1	56.93	11.65	4.88	8
RAS + ILUT	2048	1	72.39	16.00	4.52	10
RAS + ILUT	8	2	30.54	8.94	3.40	5
RAS + ILUT	16	2	30.87	8.67	3.55	5
RAS + ILUT	512	2	85.60	15.35	5.56	8
RAS + ILUT	2048	2	69.15	17.26	4.00	7

More calculations are performed in this example to test the effects of overlap and the number of blocks. We find that if we keep the overlap fixed and increase the number of blocks, the number of iterations increases. For example, when ILUT is applied and the overlap is set to 0, the number of iterations increases from 7 to 21 when the block number increases. In contrast, if the number of blocks is kept fixed and the overlap increases, the number of iterations reduces. In this case, RAS has better performance. For example, when ILU(0) is applied and the number of blocks is 2048, the numbers of iterations for 0-overlapping RAS and 1-overlapping RAS are 44 and 23, respectively.

For ILU(0), it still has better speedup than ILUT, where the average speedups for ILU(0) and ILUT are about 6.8 and 4, respectively. However, ILUT has better overall performance in terms of the total running time and the number of iterations.

5. Conclusion

In this paper, we have developed and tested a RAS preconditioner for the solution of triangular problems. Based on our parallel triangular solver, a parallel RAS preconditioner on GPU is implemented. From the numerical experiments, we can see that our GPU-based RAS preconditioner has high parallel performance and it can achieve a maximum speedup of about 10.

Acknowledgements

The support of Department of Chemical and Petroleum Engineering, University of Calgary and Reservoir Simulation Group is gratefully acknowledged. The research is partly supported by NSERC/AIEE/Foundation CMG and AITF Chairs.

References

[1] S. Balay, W. Gropp, L. McInnes and B. Smith, The Portable, Extensible Toolkit for Scientific Computing, version 2.0.13, 1996.
[2] N. Bell and M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation, 2008.
[3] N. Bell and M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, Proc. Supercomputing, November 2009, pp. 1-11.
[4] X.-C. Cai and M. Sarkis, A restricted additive Schwarz preconditioner for general sparse linear systems, SIAM J. Sci. Comput., 21, 1999, pp. 792-797.
[5] J. Cao and J. Sun, An Efficient and Effective Nonlinear Solver In A Parallel Software for Large Scale Petroleum Reservoir Simulation, International Journal of Numerical Analysis and Modelling, 2(2005), pp. 15–27.
[6] Z. Chen, G. Huan, and Y. Ma, Computational methods for multiphase flows in porous media, in the Computational Science and Engineering Series, Vol. 2, SIAM, Philadelphia, 2006.
[7] Z. Chen and Y. Zhang, Development, analysis and numerical tests of a compositional reservoir simulator, International Journal of Numerical Analysis and Modeling 4 (2008), pp. 86-100.
[8] Z. Chen and Y. Zhang, Well flow models for various numerical methods, International Journal of Numerical Analysis and Modeling 6 (2009), pp. 375-388.
[9] T. A. Davis, University of Florida sparse matrix collection, NA digest, 1994.
[10] R. Grimes, D. Kincaid, and D. Young, ITPACK 2.0 User's Guide, Technical Report CNA-150, Center for Numerical Analysis, University of Texas, August 1979.
[11] G. Haase, M. Liebmann, C. C. Douglas and G. Plank, A Parallel Algebraic Multigrid Solver on Graphics Processing Units, High Performance Computing and Applications, 2010, pp. 38-47.
[12] G. Karypis and V. Kumar, A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on Scientific Computing, 20(1), 1999, pp. 359-392.
[13] H. Klie, H. Sudan, R. Li, and Y. Saad, Exploiting capabilities of many core platforms in reservoir simulation, SPE RSS Reservoir Simulation Symposium, 21-23 February 2011
[14] R. Li and Y. Saad, GPU-accelerated preconditioned iterative linear solvers, Technical Report umsi-2010-112, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2010.
[15] H. Liu, S. Yu, Z. Chen, B. Hsieh and L. Shao, Parallel Preconditioners for Reservoir Simulation on GPU, SPE Latin American and Caribbean Petroleum Engineering Conference held in Mexico City, Mexico, 16-18 April 2012, SPE 152811-PP.
[16] NVIDIA Corporation, Nvidia CUDA Programming Guide (version 3.2), 2010.
[17] NVIDIA Corporation, CUDA C Best Practices Guide (version 3.2), 2010.
[18] NVIDIA Corporation, CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph, http://code.google.com/p/cusp-library/
[19] Y. Saad, Iterative methods for sparse linear systems (2nd edition), SIAM, 2003.
[20] L. Zhang, A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection, Numer. Math.: Theory, Methods and Applications, 2(2009), pp. 65–89.

Center for Computational Geosciences, Xi'an Jiaotong University, Xi'an 710049, China.

Department of Chemical and Petroleum Engineering, University of Calgary, Calgary, AB, Canada, T2N 1N4.

E-mail: `hui.j.liu@ucalgary.ca, yusong0926@gmail.com, zhachen@ucalgary.ca`
E-mail: `bhsieh@ucalgary.ca, jedyfun@gmail.com`
URL: `http://schulich.ucalgary.ca/chemical/JohnChen`