INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING, *SERIES B*
Volume 5, Number 1-2, Pages 136–146

ACCELERATING PRECONDITIONED ITERATIVE LINEAR SOLVERS ON GPU

HUI LIU, ZHANGXIN CHEN AND BO YANG

Abstract. Linear systems are required to solve in many scientific applications and the solution of these systems often dominates the total running time. In this paper, we introduce our work on developing parallel linear solvers and preconditioners for solving large sparse linear systems using NVIDIA GPUs. We develop a new sparse matrix-vector multiplication kernel and a sparse BLAS library for GPUs. Based on the BLAS library, several Krylov subspace linear solvers, and algebraic multi-grid (AMG) solvers and commonly used preconditioners are developed, including GMRES, CG, BICGSTAB, ORTHOMIN, classical AMG solver, polynomial preconditioner, ILU(k) and ILUT preconditioner, and domain decomposition preconditioner. Numerical experiments show that these linear solvers and preconditioners are efficient for solving the large linear systems.

Key words. Krylov subspace solver, algebraic multi-grid solver, parallel preconditioner, GPU computing, sparse matrix-vector multiplication, HEC

1. Introduction

Linear and nonlinear solvers play an important role in scientific computing areas. In many scientific applications, the solution of systems of linear algebraic equations dominates the whole simulation time. The black oil simulator [1], for example, may run for weeks or even months depending on the problem size. In this simulator, iterative linear solvers may take up to 98% of the whole simulation time. Therefore, the development of fast and accurate linear solvers and preconditioners are essential to many applications.

Efficient iterative linear solvers, preconditioners and parallel computing techniques have been developed to accelerate the solution of linear systems. Saad developed the GMRES solver, a general solver for unsymmetric linear system [22, 23] and Vinsome designed the ORTHOMIN solver, which was originally developed for reservoir simulation [2], for example. Commonly used preconditioners were also developed, such as Incomplete LU (ILU) factorization, domain decomposition and algebraic multigrid preconditioners [22, 23]. GPUs (Graphics Processing Unit) are usually used for display, since each pixel can be processed simultaneously. GPUs are designed in such a way that they can manipulate data in parallel. They have parallel architectures and their memory speed is very high [5, 6, 7]. In general, GPUs are ten to forty times faster than general CPUs (Central Processing Units) [5, 6, 7], which makes them proper devices for parallel computing, especially for developing fast linear solvers. The architectures of GPUs are different from those of CPUs, and therefore, new algorithms for GPUs should be designed to match the architectures of GPUs. Efforts have been made to utilize GPUs' performance [5, 6, 7, 3, 4, 27, 8, 10, 11]. Bell and Garland investigated different kinds of matrix formats and SpMV algorithms in [3, 4], in which the HYB format matrix and the corresponding SpMV algorithm was also designed. Naumov from NVIDIA developed a fast triangular solver for GPU and the solver was over two times faster

Received by the editors January 12, 2014 and, in revised form, March 20, 2014.
2000 *Mathematics Subject Classification.* 65F30, 65F50, 65Y05, 65Y10, 68W10.
This research was supported by NSERC/AIEE/Foundation CMG and AITF Chairs.

than CPU-based triangular solver. Saad et al. used JAD matrix and developed an efficient sparse matrix-vector multiplication (SpMV) kernel for GPU, and based on the kernel, several Krylov solvers and ILU preconditioner were implemented on GPU. Haase et al. developed a parallel AMG solvers using a GPU cluster [12]. Researchers from NVIDIA also developed an AMG solver on GPU [9, 13]. The setup and solving phases were both run on GPU, which made their AMG very fast. Chen et al. from University of Calgary designed a new matrix format HEC (Hybrid of Ell and CSR), fast SpMV kernel [11], parallel triangular solver [10], parallel preconditioners [8] and the GPU solvers have been applied to reservoir simulation [14, 17]. More details can be read from [5, 6, 7, 3, 4, 27, 8, 10, 11, 15, 24, 28].

In this paper, we introduce our work on developing a general purpose GPU-based parallel iterative linear solver package. We design a flexible matrix format for GPU and its corresponding SpMV algorithm [11]. The matrix is also a good choice for parallel triangular solver [28]. Based on this SpMV and other matrix-vector operations, several Krylov subspace solvers are implemented. For symmetric positive definite matrices, AMG solvers are the most effecient methods, which project a low frequency error to a coarser grid, and convert it to a high frequency error. In this case, the resulting high frequency error is easy to converge again. The convergence rate of these AMG solvers is optimal [18, 19, 20, 21] in terms of the number of iterations. We implement a classical AMG solver, which also serves as a preconditioner. Other commonly used preconditioners are also implemented, including ILU(k), ILUT(p, tol), RAS (Restricted Additive Schwarz method), polynomial and block ILU(k), block ILUT(p, tol). Numerical experiments show that our linear solvers and preconditioners are over ten times faster than CPU-based solvers and preconditioners.

The layout of the paper is as follows. In §2, GPU computing will be introduced briefly. In §3, our work on linear solvers will be proposed. IN §4, parallel preconditioners are presented. Then numerical experiment will be presented in §5.

2. GPU Computing

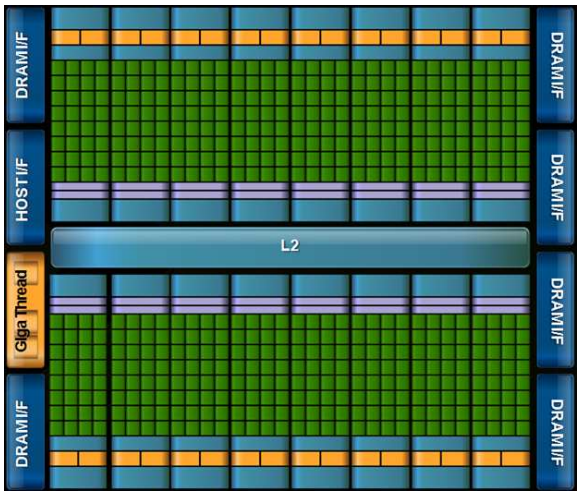

Figure 1: NVIDIA Fermi Architecture.

The architectures of GPUs are parallel [6, 7]. The NVIDIA Fermi GPU, as shown in Fig. 1 as an example, has 16 SMs (Streaming Multi-processor), and each SM has 32 SPs (Streaming Processor). This GPU has 512 processors shown as green blocks while a normal CPU has only 2, 4, 6 or 8 cores. The NVIDIA Tesla C2050 has a peak performance of 1030G FLOPS in single precision and a peak performance of 515G FLOPS in double precision, which are around 10 times faster than that of CPUs. Each SM has its own L1 cache, shared memory and register. They share L2 cache, constant memory, texture memory and global memory. The global memory stores most of data, and is used to communicate with CPUs. Its memory speed is around 150 GB/s while the memory speed of CPUs is around 15 GB/s. The GPU memory is also about 10 times faster than the CPU memory.

NVIDIA provides CUDA Toolkit [6, 7] to help users develop high performance programs. CUDA applications consist of two parts: a host part and a device part. The host program consists of pure CPU codes running on CPUs. The device program is executed on a GPU device. A device program is called a kernel function, which employs a SPMT model (Single Program Multiple Thread) [6, 7], generating a large number of threads and executing the same kernel function in parallel on streaming processors of GPUs. This bunch of threads is organized into a grid of thread blocks. Threads in the same block are able to cooperate with each other by barrier synchronization and shared memory. In addition, threads in different blocks cannot communicate with each other directly due to efficiency considerations. However, they can communicate with each other through global memory using two kernel functions.

3. GPU-based Parallel Linear Solvers

In this section, we will present our work on developing SpMV kernel, Krylov subspaces solvers and algebraic multi-grid solver.

3.1. Sparse Matrix-Vector Multiplication. For the sake of completeness, the SpMV algorithm we developed is introduced here [11]. The pattern of memory access of NVIDIA GPUs is coalesced, which means that if data access is arranged properly, threads in a grid block read and write data using only one or a few accesses. In this case, the global memory access is optimized and memory speed is highest.

For our general matrix formats, such as CSR, CSC and DIA [22, 23, 5], when we develop sparse matrix-vector multiplication kernel, it is hard to obtain maximal float point performance and memory speed. NVIDIA developed a hybrid matrix format HYB [3] and Saad et al designed the JAD matrix format [5] to improve the performance of SpMV. In this paper, we develop a different matrix format called HEC shown in Fig. 2, which is friendly to ILU-like preconditioners. The HEC format consists of two parts: ELL part and CSR part. The ELL is very regular and each row has the same length. Therefore, the memory access is coalesced and memory access speed is high. The CSR part stores the irregular part of our matrix. When designing the SpMV algorithm for GPU, we employ one thread for one row. Pseudo-codes for this is straightforward as shown in Algorithm 1.

3.2. Krylov Subspace Solvers. From references [22, 23], iterative linear solvers consist of matrix-vector multiplication operations and vector operations. These common operations include,

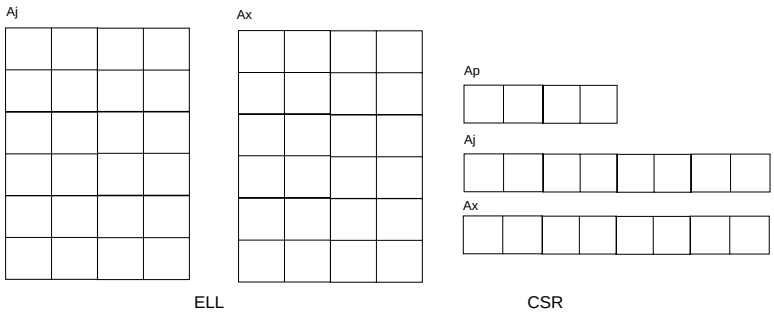

Figure 2: HEC Matrix Foramt.

Algorithm 1 Sparse Matrix-Vector Multiplication, $y = Ax$

```
1: for i = 1: n do                    ▷ ELL, Use one GPU kernel to deal with this loop
2:     the i-th thread calculate the i-th row of ELL matrix;    ▷ Use one thread
3: end for
4:
5: for i = 1: n do                    ▷ CSR, Use one GPU kernel to deal with this loop
6:     the i-th thread calculate the i-th row of CSR matrix;    ▷ Use one thread
7: end for
```

$$y = \alpha Ax + \beta y, \quad \alpha, \beta \in R, \tag{1}$$

$$z = \alpha Ax + \beta y, \quad \alpha, \beta \in R, \tag{2}$$

$$y = \alpha x + \beta y, \quad \alpha, \beta \in R, \tag{3}$$

$$z = \alpha x + \beta y, \quad \alpha, \beta \in R, \tag{4}$$

$$\alpha = \langle x, y \rangle, \tag{5}$$

where A is a matrix, x, y and z are vectors, α and β are real numbers, and $\langle \cdot, \cdot \rangle$ is the scalar product. If all these operations are implemented, Krylov [22, 23] subspace solvers can be implemented straightforwardly. We implement a GPU BLAS library, which contains commonly used BLAS 1 and BLAS 2 subroutines. With the help of the BLAS library, we implement several Krylov linear solvers, including CG, GMRES, BICGSTAB, CGS, ORTHOMIN, ORTHODIR and GCR [22, 23].

3.3. Algebraic Multi-Grid Solver. For linear system $Ax = b$, A is a sparse positive definite matrix ($A \in R^{nxn}$). This linear system may be derived from the discretization of a pressure equation or parabolic equations by using the finite element method, the finite difference method or the finite volume method.

The structure of the standard V-cycle AMG solver is shown in Fig. 3 [15], which has $L + 1$ levels and the original linear system $Ax = b$ stands for the finest level ($l = 0$) and level L is the coarsest level. An AMG solver consists of two phases, a setup phase and a solution phase. In the setup phase, the hierarchical system shown in Fig. 3 is assembled. On each level l except level L, a coarse grid is chosen, and the coarser grid block size should be much larger than the current grid block size such that the linear system on the coarse grid is easy to solve, and the error on it should approximate the error on the current fine grid. Then a restriction operator

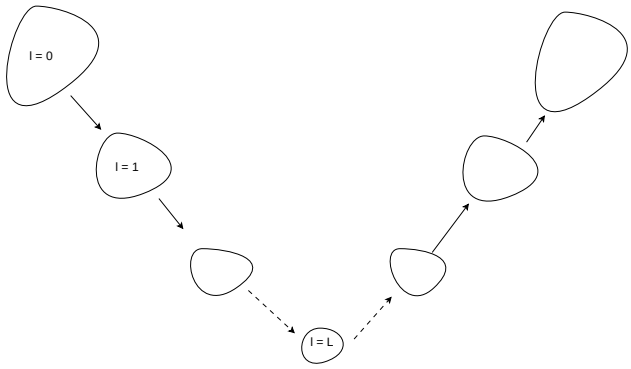

Figure 3: Structure of AMG solver.

R and an interpolation (prolongation) operator P are determined. The restriction operator projects an error from a finer grid onto a coarser grid and converts a low frequency error to a high frequency error while the interpolation operator transfers a solution on a coarser grid to that on a finer grid. In general, the restriction operator R is the transpose of the interpolation (prolongation) operator P; i.e., $R = P^T$. The setup phase on each level l $(0 \leq l < L)$ is formulated in Algorithm 2.

Algorithm 2 AMG setup Algorithm

1: Calculate strength matrix S.
2: Choose coarsening nodes set ω_{l+1} according to strength matrix S, such that $\omega_{l+1} \subset \omega_l$.
3: Calculate prolongation operator P_l.
4: Derive restriction operator $R_l = P_l^T$.
5: Calculate coarse matrix A_{l+1}: $A_{l+1} = R_l \times A_l \times P_l$.
6: Choose pre-smoother S_l and post-smoother T_l.

The coarsening algorithms we use here are the RS algorithm introduced by Ruge and Stüben [18, 19] and the CLJP algorithm introduced by Cleary, Luby, Jones and Plassmann.

The prolongation operators are the classical one proposed by Ruge and Stüben and the one introduced in Haase's paper [12]. The second one is simpler and has lower complexity compared to the classical prolongation operator. Let P be the second prolongation operator to be calculated; we have

$$P_{i,j} = \begin{cases} 1 & i \text{ is a coarening node,} \\ 0 & i \text{ is a fine node and is weakly connected with j,} \\ 1/n_i & i \text{ is a fine node and is strongly connected with j,} \end{cases} \tag{6}$$

where $P_{i,j}$ is the (i,j)-th element of operator P and n_i is the number of coarsening nodes connected with fine node i.

For a given matrix A, the right-hand side b and the initial solution x, the pre-smoother S_l and the post-smoother T_l have the same form so that

$$x^{m+1} = x^m + \alpha M^{-1}(b - Ax^m) = x^m + \alpha M^{-1} r. \tag{7}$$

The difference is how to choose matrix M and coefficient α, where α is positive and often equals to 1. If M is chosen as the diagonal matrix of A, we have a Jacobi

smoother, and if α is less than 1, we have a damped Jacobi. If M is the block diagonal part of A, we have a block Jacobi smoother. In our solver package, we have implemented the Jacobi, damped Jacobi, block Jacobi, weighted Jacobi, Gauss-Seidel, block Gauss-Seidel, approximate inverse, domain decomposition, polynomial and hybrid smoothers. For the block Jacobi, Gauss-Seidel, block Gauss-Seidel and domain decomposition smoothers [25, 16], triangular systems are required to solve. Parallel GPU-based triangular solvers [8] are applied here.

The solution phase of AMG is recursive and is formulated in Algorithm 3, which shows one iteration of AMG. On each level l, the algorithm requires A_l, R_l, P_l, S_l and T_l. A_l, R_l and P_l are matrices. The pre-smoother S_l and post-smoother T_l are typically damped Jacobi, Gauss-Seidel, block Gauss-Seidel and domain decomposition smoothers. We assemble the setup phase on CPU solve the system on GPU.

Algorithm 3 AMG Solution Algorithm: mg_solve(l)

Require: b_l, x_l, A_l, R_l, P_l, S_l, T_l, $0 \leq l < L$

$b_0 = b$
if $(l < L)$ **then**
  $x_l = S_l(x_l, A_l, b_l)$ ▷ Pre-smoothing
  $r = b_l - A_l x_l$
  $b_{r+1} = R_l r$ ▷ Restriction
  mg_solve$(l+1)$ ▷ Next level
  $x_l = x_l + P_l x_{l+1}$ ▷ Prolongation
  $x_l = T_l(x_l, A_l, b_l)$ ▷ Post-smoothing
else
  $x_l = A_l^{-1} b_l$
end if
x $= x_0$

4. Parallel Preconditioners

It is well known that preconditioners are essential to the speed up and convergence of iterative linear solvers. If a preconditioner is chosen properly, the solution time of linear systems can be reduced dramatically. We implement polynomial preconditioner, Incomplete LU (ILU) factorization preconditioner, block ILU factorization and domain decomposition preconditioner [22, 23, 25].

For any nonsingular matrix A, we write A as $A = I - B$. If the spectral radium of B is less than one, then we have the following Neumann polynomial:

$$A^{-1} = I + B^1 + B^2 + B^3 + B^4 + \cdots . \tag{8}$$

If we define $P_k(A)$ as follows:

$$P_k(A) = I + B^1 + B^2 + B^3 + B^4 + \cdots B^k, \tag{9}$$

then $P_k(A)$ is a kth-order Neumann polynomial preconditioner. The solution of a preconditioned linear system equals to SpMV.

For any given square non-singular matrix A, the ILU factorization computes a sparse lower triangular matrix L and a sparse upper triangular matrix U. If no fill-in is allowed, we obtain the so-called ILU(0) preconditioner. If fill-ins are

allowed and computed by non-zero pattern using method introduced in [23], ILU(k) can be obtained. Another method is ILUT, which drops entries based on the numerical values of the fill-in elements [23, 5]. When block Jacobi method is choose as preconditioner, each submatrix is solveed by ILU(k) or ILUT, then block ILU(k) and block ILUT are implemented. When the number of blocks is set to one, they are usual ILU(k) and ILUT, respectively.

Domain decomposition methods use the principle of divide-and-conquer. These methods have been primarily developed for solving Partial Differential Equations (PDEs) over regions in two or three dimensions [23] and for parallel computing. Cai et al. developed the restricted additive Schwarz method for a general sparse matrix [25], which reduces communication and is much faster than the classical additive Schwarz preconditioner. Its basic idea is to restrict the original large matrix to some smaller matrices and then to solve these smaller matrices simultaneously. In this paper, we treat the original matrix as an undirected graph and this graph is partitioned by METIS [16]. The subdomain can be extended according to the topology of the graph. Then each smaller problem can be solved by ILU(k) or ILUT.

For ILU preconditioners and RAS preconditioner, the following triangular systems need to be solved:

$$LUx = b \Leftrightarrow Ly = b, \quad Ux = y, \tag{10}$$

where L and U are a lower triangular matrix and an upper triangular matrix, respectively, b is the right-hand side vector and x is the unknown to be solved for. The lower triangular problem, $Ly = b$, is solved first, and then, by solving the upper triangular problem, $Ux = y$, we can obtain the result x. For GPU computing, the parallel triangular solvers are based on the level schedule method [23], and details can be read from [23, 28, 8].

5. Numerical Experiments

Numerical experiments are performed on our workstation with Intel Xeon X5570 CPUs and NVIDIA Tesla C2050/C2070 GPUs. The operating system is CentOS 6.2 X86_64 with CUDA Toolkit 4.0 and GCC 4.4. All CPU codes are compiled with -O3 option. The type of float point number is double. Performance of our SpMV kernel can be found [11].

Example 1. *The matrix is from SPE 10* [1]*. Its dimension is 2,188,851 and the number of non-zeros is 29,915,573. Three solvers are tested without any preconditioner, and the number of iteration is fixed at 20. Data is shown in Table 1.*

Table 1: Performance of solvers without preconditioner

Solver	CPU (s)	GPU (s)	Speedup
BICGSTAB	3.27	0.31	9.95
ORTHOMIN(20)	5.95	0.52	10.61
GMRES(20)	60.39	5.61	10.72
GMRES(40)	178.08	17.01	10.45

This example is for testing the framework of our package and the performance of the BLAS library. From Table 1, we can see that our linear solvers are efficient and they are around 10 times faster than the CPU version solvers.

Example 2. *The matrix used here is from a three-dimensional Poisson equation. Its dimension is 1,000,000 and the number of non-zeros is 6,940,000. Performance data is collected in Table 2.*

Table 2: Performance of the matrix from the Poisson equation

Preconditioner	Parameters	CPU (s)	GPU (s)	Speedup
BILUK	1 block	16.31	2.15	7.55
BILUK	16 blocks	20.12	2.53	7.93
BILUK	128 blocks	20.2	2.39	8.44
BILUK	512 blocks	22.76	2.70	8.39
BILUT	1 block	7.11	1.67	4.24
BILUT	16 blocks	14.85	2.38	6.19
BILUT	128 blocks	14.56	2.30	6.30
BILUT	512 blocks	18.59	2.71	6.83
RAS	256 blocks, overlap = 1	18.70	2.16	8.60
RAS	2048 blocks, overlap = 1	22.03	2.35	9.32
RAS	256 blocks, overlap = 2	20.27	2.18	9.24
RAS	2048 blocks, overlap = 2	22.23	2.66	8.30

In this example, the solver is GMRES(20). Three different preconditioners are tested, which are block ILU(0), block ILUT and domain decomposition, noted by BILUK, BILUT and RAS. For the BILUK, when the number of blocks is one, it is ILU(0). Though it is highly sequential, we can still speed ILU(0) around 7.5 times faster. When increasing the number of blocks, our solver has better speedup. The average speedup for this matrix is about 8.

BILUT is a special preconditioner since it is obtained according to the values of L and U; sometimes, the sparse patterns of L and U are more irregular than those in BILUK [8]. From Table 2, we can see that the speedups are a little lower compared to BILUK. However, BILUT reflects the real data dependence, and its performance is better than BILUK in general. This is demonstrated by Table 2. The CPU version BILUT always takes less time than the CPU BILUK. But, for the GPU versions, their performance is similar.

Here the RAS means restricted additive schwarz preconditioner proposed by Cai et al [25]. The RAS preconditioner is always stable. It also has better speedup than BILUK and BILUT, which is demonstrated by Table 2. The average speedup is around 9.

Example 3. *The matrix tested is from a three-dimensional elliptic partial differential equation. Its grid size is 100x100x100. The dimension of the matrix is 1,000,000 and the number of non-zeros is 6,940,000. The RS coarsening strategy and damped Jacobi smoother are applied. The AMG solver has 8 levels. The termination criterium is 1e-6* [15].

The performance data is shown in Fig. 4. Both CPU and GPU solvers terminate in 21 iterations. The GPU takes 1.127s to solve the linear system shown in green color while the CPU solver takes 10.8s to solve it shown in red color. The GPU-based parallel AMG solver is 9.58 times faster than the CPU-based AMG solver.

Example 4. *The matrix tested is also from a three-dimensional elliptic partial differential equation. Its grid size is 130x130x130. The dimension of the matrix is*

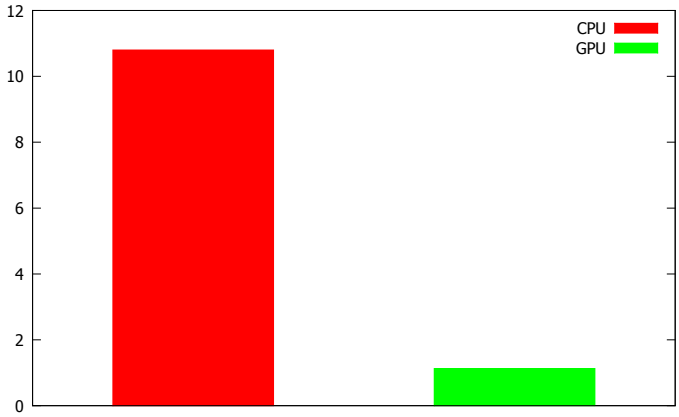

Figure 4: Running time of Example 3

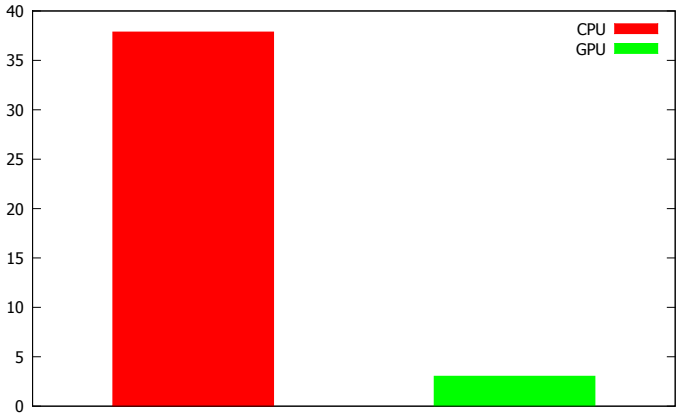

Figure 5: Running time of Example 4

2,197,000 and the number of non-zeros is 15,277,600. The RS coarsening strategy and weighted Jacobi smoother are applied. The AMG solver has 8 levels. The termination criterium is 1e-6 [15].

The performance of example 4 is shown in Fig. 5. AMG solvers terminate in 25 iterations. The GPU-based AMG solver takes 3.02s shown in green color while the CPU-based AMG solver takes 37.86s. The speedup of the GPU-based AMG solver for this matrix is around 12.47. This parallel solver is much more efficient than the CPU-based solver.

6. Conclusion

In this paper, we present our GPU-based parallel iterative linear solvers and preconditioners. A GPU BLAS library, including SpMV and vector operations, several Krylov subspace linear solvers,classical AMG solver and commonly used preconditioners are implemented. The numerical experiments show that our solvers

and preconditioners are efficient. From these experiments, we can conclude that our GPU-based linear solvers are around ten times faster than the corresponding CPU-based linear solvers.

Acknowledgments

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] 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).

[2] P.K.W Vinsome, *an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations*, *SPE Symposium on Numerical Simulation of Reservoir Performance* (Los Angeles, California, 1976).

[3] N. Bell and M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation, 2008.

[4] N. and M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, Proc. Supercomputing, November 2009, 1-11.

[5] 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).

[6] NVIDIA Corporation, *Nvidia CUDA Programming Guide (version 3.2)* (2010).

[7] NVIDIA Corporation, *CUDA C Best Practices Guide (version 3.2)* (2010).

[8] Hui Liu, Song Yu, Zhangxin Chen, Ben Hsieh, and Lei Shao, *Parallel Preconditioners for Reservoir Simulation on GPU*, *SPE Latin America and Caribbean Petroleum Engineering Conference* (Mexico City, Mexico, 2012).

[9] NVIDIA Corporation, CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph, http://code.google.com/p/cusp-library/

[10] Song Yu, Hui Liu, Zhangxin Chen, Ben Hsieh, and Lei Shao, *GPU-based Parallel Reservoir Simulation for Large-scale Simulation Problems*, *SPE Europec/EAGE Annual Conference* (Copenhagen, Denmark, 2012).

[11] Hui Liu, Song Yu, Zhangxin Chen, Ben Hsieh and Lei Shao, *Sparse Matrix-Vector Multiplication on NVIDIA GPU*, International Journal of Numerical Analysis and Modeling, Series B, Volume 3, No. 2 (2012).

[12] G. Haase, M. Liebmann, C. C. Douglas and G. Plank, A Parallel Algebraic Multigrid Solver on Graphics Processing Units, High performance computing annnd applications, 2010, 38-47.

[13] Nathan Bell, Steven Dalton and Luke Olson, Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods, NVIDIA Technical Report NVR-2011-002, June 2011

[14] Zhangxin Chen, Hui Liu, Song Yu, Ben Hsieh, Lei Shao, Reservoir Simulation on NVIDIA Tesla GPUs, The Eighth International Conference on Scientific Computing and Applications, University of Nevada, Las Vegas, April, 2012.

[15] Hui Liu, Song Yu and Zhangxin Chen, Development of Algebraic Multigrid Solvers Using GPUs, SPE Reservoir Simulation Symposium, Feb 2013, Houston, USA.

[16] 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.

[17] Zhangxin Chen, Hui Liu, Song Yu, Ben Hsieh and Lei Shao, GPU-based parallel reservoir simulators, 21st International Conference on Domain Decomposition Methods, 2012, France.

[18] K. Stüben, A review of algebraic multigrid, Journal of Computational and Applied Mathematics Volume 128, Issues 1-2, 2001, 281–309.

[19] J.W. Ruge and K. Stüben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, Vol. 5, SIAM, Philadelphia, 1986.

[20] A. Brandt, S.F. McCormick and J. Ruge, Algebraic multigrid (AMG) for sparse matrix equations D.J. Evans (Ed.), Sparsity and its Applications, Cambridge University Press, Cambridge, 1984, 257–284.

[21] C. Wagner, Introduction to Algebraic Multigrid, Course notes of an algebraic multigrid course at the University of Heidelberg in the Wintersemester, 1999.

[22] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. Van der Vorst , Templates for the solution of linear systems: building blocks for iterative methods, 2nd Edition, SIAM, 1994.
[23] Y. Saad, Iterative methods for sparse linear systems (2nd edition), SIAM, 2003.
[24] Muthu Manikandan Baskaran and Rajesh Bordawekar, Optimizeing Sparse Matrix-Vector Multiplication on GPUs, In Ninth SIAM Conference on Parallel Processing for Scientific Computing, 2008.
[25] 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.
[26] 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.
[27] 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
[28] Z. Chen, H. Liu and B. Yang, Parallel triangular solvers on GPU, to appear.
[29] T. A. Davis, University of Florida sparse matrix collection, NA digest, 1994.

Center for Computational Geosciences, College of Mathematics and Statistics, Xi'an Jiaotong University, Xi'an 710049, China.

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

E-mail: hui.j.liu@ucalgary.ca, zhachen@ucalgary.ca, yang6@ucalgary.ca
URL: http://schulich.ucalgary.ca/chemical/JohnChen