A Parallel Preconditioned Bi-Conjugate Gradient Stabilized Solver for the Poisson Problem

Zhao Ning[a,b]
a.College of Geophysics,ChengDu University of Technology ,ChengDu,China
b.School of Computer Science and Technology,Henan Polytechnic University
Email: zhaoning@hpu.edu.cn

Wang XuBen
Key Lab of Earth Exploration&Information Techniques of Ministry of Education .ChengDu University of Technology ,ChengDu,China
Email: wxb@cdut.edu.cn

***Abstract*—We present a parallel Preconditioned Bi-Conjugate Gradient Stabilized(BICGstab) solver for the Poisson problem. Given a real, nosymmetric and positive definite coefficient matrix , the parallized Preconditioned BICGstab -solver is able to find a solution for that system by exploiting the massive compute power of todays GPUs.Comparing sequential CPU implementations and that algorithm.we achieve a speed up from 8 to 10 depending on the dimension of the coefficient matrix. Additionally the concept of preconditioners to decrease the time to find a solution is evaluated using the AINV method.**

***Index Terms*—sparse matrix solver, Bi-Conjugate Gradient Stabilized, ELLPACK-R, NVIDIA CUDA, AINV precondition**

I. INTRODUCTION

Compared to the classical CPU environments, The graphics processors play an important role in parallel and distributed systems. The floating point processing provides general purpose computations on GPGPU. For this reason,numerical calculation can be accelerated as long as the algorithms well to the characteristics of the hardware. In communication-and bandwidth-limited problems, The algorithm for solving large-scale sparse group of linear equations causes negative speedups. Our job aim to the Poisson problem that arises in a large range of disciplines like gravity,magnetic,electrical,and seismic [1]. The numerical calculation result of mass data discretization must be processed by the system, especially in three dimension compute. Thus, Iterative solution usually instead of direct solution.

The iterative methods are more amenable to parallelism and therefore can be used to solve larger problems. Currently, the most popular iterative schemes belong to the Krylov subspace family of methods. They include Preconditioned Bi-Conjugate Gradient Stabilized (BiCGSTAB) and Preconditioned Conjugate Gradient (CG) iterative methods for nonsymmetric and symmetric positive definite (s.p.d.) linear systems.In practice, we often use a variety of preconditioning techniques to improve the convergence of iterative method. In this paper we focus on the Approximate Inverse(AINV) preconditioning for the numerical simulation [2].

The preconditioned Bi-Conjugate Gradient Stabilized was introduced in as an efficient method to solve linear equation systems with real, symmetric and positive definite coefficient matrices. With the appearance of programmable graphics hardware a cheap way for getting massive parallel processors to the masses became possible. Therefore, general people can also take advantages from parallel algorithms. When analysing the conjugate gradient algorithm one will recognize that serveral operations within one iteration of the algorithm can be computed in parallel. In a parallel implementation was introduced using GLSL shader programs. The necessary data for the computation was stored in textures and the algorithm was implemented in a pixel shader. Due to the general purpose GPU paradigm encoding of data in textures got superfluous because new technologies, like the NVIDIA CUDA architecture allow programming the parallel processors using an extended version of the C programming language.If one wants to reach a time efficient implementation of the Preconditioned Bi-Conjugate Gradient Stabilized. it is essential to have an efficient way to compute sparse matrix vector products. In an efficient storage for sparse matrices the ELLPACK format was suggested, which was extended to the ELLPACK-R format in especially for the use on GPUs. For time efficient solution of linear systems it is not enough to have efficient data structures and a lot of compute power: There exist extensions to the conjugate gradient algorithm which use a preconditioning matrix which is applied to the system matrix to decrease its condition number. In this paper a sequential version of such preconditioner is evaluated using the AINV method described in .

Corresponding author: Zhao Ning (zhaoning@hpu.edu.cn)

II. BASICS

doi:10.4304/jcp.7.12.3088-3095

A. The Poisson Equation and its Applications

The Poisson equation is a second-order partial differential equation that arises in a large range of electrical problems[3]. Its common form is

(1) $\Delta\phi = f$

where Δ denotes the Laplacian. In three-dimensional Cartesian coordinates, the Poisson equation can express as:

(2) $\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=f$

Analytic Method for the Poisson equation are usually not operate. A general approach is to discretize the domain into a finite number of grid points.Discretization schemes for partial differential equations such as the Finite Difference, the Finite Element, or the Finite Volume scheme ultimately lead to the need for the solution of a large system of linear equations. Since the coupling between the equations is usually rather weak, the system matrix is very sparse, allowing for millions of unknowns on average workstations[4]. For such huge systems, iterative solution methods are typically employed, where the convergence rate depends on the condition number of the iteration matrix. Consequently,in practical applications it is of interest to keep the condition number low, for which so-called preconditioners are employed. Formally, one way of interpreting the action of a preconditioner is to multiply the original system ,as in (1) .

$$A \bullet X = B \quad (1)$$

For example, An Finite Difference Frequency Domain method solves Maxwell's equations,where A is a nosymmetric complex matrix. The matrix A depends onthe conductivities and grid spacings, b is a vector containing source terms associated withthe boundary conditions, and x is the unknown complex vector of the electric field components throughout the model.Nowadays, the system is commonly solved iteratively by a preconditioned Krylov method . In this respect, only two aspects are important, first, how accurate the system $A \bullet X = B$, and second, how well-preconditioned the system matrix A is .

B. The Preconditioned Bi-Conjugate Gradient Stabilized

In last several years, a number of Krylov subspace algorithms have been proposed for processing linear systems.These methods proceed in an iterative manner to minimize,as in (2).

$$r = Ax-b \quad (2)$$

The error is not reduced at each iteration,and thus the convergence is generally erratic,these techniques have proven successful in reducing the error to a predetermined level within an acceptable number of iterations.The most largely used of the techniques is the preconditioned Biconjugate Gradient Stabilized (BiCGSTAB)method for electromagnetic modeling by Sarkar and recently for magnetotelluric modeling by smith[5].The pseudocode for the preconditioned BiCGSTAB iterative method is shown in Alg. 1.

1. $r_0 = b - Ax_0$
2. Choose an arbitrary vector r_0 such that $(r_0, r_0) \neq 0$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $v_0 = p_0 = 0$
5. For $i = 1, 2, 3, \ldots$
   1. $\rho_i = (r_0, r_{i-1})$
   2. $\beta = (\rho_i / \rho_{i-1})(\alpha / \omega_{i-1})$
   3. $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
   4. $y = K^{-1} p_i$
   5. $v_i = Ay$
   6. $\alpha = \rho_i / (r_0, v_i)$
   7. $s = r_{i-1} - \alpha v_i$
   8. $z = K^{-1} s$
   9. $t = Az$
   10. $\omega_i = (K^{-1}t, K^{-1}s) / (K^{-1}t, K^{-1}t)$
   11. $x_i = x_{i-1} + \alpha y + \omega_i z$
   12. If x_i is accurate enoughthen quit
   13. $r_i = s - \omega_i t$

Algorithm 1. Preconditioned BiCGSTAB

C. AINV Preconditioning Techniques

One approach to advance stability and acceleration is the preconditioning of the coefficient matrix A. The cgalgorithm converged much faster for coefficient matrices with smaller condition number.

The computational complexity of preconditions is far lower than for handling A^{-1} process.It is to transform Ax = b into an equivalent system, and the coefficient matrix condition number is far less than the cond (A) linear algebraic equations.The equivalent system of linear equations select the appropriate commonly used iterative method,. At this time, due to the coefficient matrix of the condition number is small[6],The iteration method of convergence fastly reach the purpose of accelerated iteration.The precondition of Incomplete Factorization or SSOR are well suitable for cpu processing and can greatly reduce the computating time. However, these preconditioners are unfit for GPU architectures. The Approximate Inverse (AINV), which is to find an $M^{-1} \approx A^{-1}$ directly without going the detour through an approximation of A.,is a good choice.

Based on the matrix decomposition, sparse approximate inverse with Krylov subspace methods has many advantages,If A is symmetric positive definite matrix,the precondition of symmetric positive definite can keep coefficient matrix. After pretreatment of linear system can use the CG method to solve. Compared to other approximate inverse algorithm, it has a small amount of calculation and the parameters,so that to reduce the uncertainty of the system.

AINV algorithm provided by Benzi et al.In order to keep sparsity ,we consider a program based on a direct method of matrix inversion. This lead to a factorized sparse $G \approx A^{-1}$.

Our preconditioner is based on an algorithm which computes two sets of vectors $\{Z_i\}_{i=1}^{n}$, $\{w_i\}_{i=1}^{n}$ which are

A-biconjugate, such that $w_i^T A z_j = 0$ if $i \neq j$. Given a matrix $A \in R^{n \times n}$,Between the problem of inverting A and that of computing two sets of A-biconjugate vectors $\{Z_i\}_{i=1}^n$, $\{w_i\}_{i=1}^n$=1 .

It computes two sets of vectors $\{Z_i\}_{i=1}^n$, $\{W_i\}_{i=1}^n$,If we introduce the matrices:

$$Z=[z_1, z_2, ..., z_n] \text{ and } W=[w_1, w_2, ..., w_n],$$

then

$$W^T AZ = D = diag(p_1, p_2, ..., p_n),$$

Where $p_i = w_i^T AZ_i \neq 0$. It follows W and Z are necessarily nonsingular and

$$A^{-1} = ZD^{-1}W^T = \sum_{i=1}^{n} \frac{z_i w_i^T}{p_i}$$

Therefore, A has an LU factorization, Z can be computed make use of a biconjugation process applied to the basis vectors $e_1, e_2, ...e_n$ .It is easily observed that $Z = U^{-1}$ and $W = L^{-T}$ where $A = LDU$ which L and U lower and upper triangular and D diagonal.If a_i^T denotes the i row of A, C_i is the i column of A, the biconjugatioin procedure to compute Z can be written as follows[7]:

① Let $z_1^0 = e_1, p_1^0 = a_{11}$
② for i=2,...,n,do
③ $z_i^0 = e_i$
④ for j=1,2,...,i-1,do
⑤ $p_i^{j-1} := a_j^T z_i^{j-1}$
⑥ $z_i^j := a_j^{j-1} - \left[\frac{p_i^{j-1}}{p_j^{j-1}}\right] z_j^{j-1}$
⑦ end do
⑧ $p_i^{i-1} := a_i^T z_i^{i-1}$
⑨ end do

Algorithm2. AINV algorithm

If p_i is a negative or zero value. Benz pointed out that $p_i := z_i^T A z_i > 0$ .However,if A is an symmetrical matrix. the resulting SAINV preconditioner is general extremely effective[8] . In general , accuracy zero is very impossible to occur , resulting in uncontrolled growth of the entries of Z and W. So as to avoiding the happening of fault, by analyzing the reason of pivots may be negative or zero value and avoiding them.

The AINV preconditioner consisting of matrix–vector products is easily parallelized on GPU computer. One possible solution is to compute the preconditioner sequentially on one processor and then to allocate factors among processors. Relative to the overall costs, the time for computing the preconditioner is negligible[9].

III. GPU ARCHITECTURE AND CUDA

A .GPU Architecture

GPU is built as a scalable array of multi-threaded Streaming Multiprocessors (SMs), each of which consists of multiple Scalar Processor (SP) cores. To manage hundreds of threads, the multiprocessors employs a Single Instruction Multiple Threads (SIMT) model. The NVIDIA GPU platform has various memory hierarchies. The types of memory can be classified as follows: (1) global memory, (2) local memory, (3) on-chip shared memory, (4) read-only cached off-chip constant memory and texture memory and (5) registers. The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory is relatively large but has a much higher latency (from 400 to 800 cycles for the NVIDIA GPUs) compared with the on-chip shared memory (1 clock cycle if there are no bank conflicts).Global memory is not cached, so it is important to follow the right access pattern to achieve good memory bandwidth[10].

Threads are organized in warps. A warp is defined as a group of 32 threads of consecutive threadIDs. A half-warp is either the first or second half of a warp. The most efficient way to use the global memory bandwidth is to coalesce the simultaneous memory accesses by threads in a half-warp into a single memory transaction. For NVIDIA GPU devices with compute capability and higher,memory access by a half-warp is coalesced as soon as the words accessed by all threads lie in the same segment of size equal to 128 bytes if all threads access 32-bit or 64-bit words. Local memory accesses are always coalesced. Alignment and coalescing techniques are explained in details in [11].

Since it is on chip, the shared memory is much faster than the global memory but we also need to pay attention to the problems of bank conflict. The size of shared memory is 16K per SM. Cached constant memory and texture memory are also beneficial for accessing reused data and data with 2D spatial locality[12].

B. CUDA Concepts

The parallelization of the algorithm is done using the NVIDIA CUDA technology. This technology makes it possible to exploit the massive compute power of todays GPUs for general purpose computing by creating kernel programs which execute in parallel.

Next the CUDA programming paradigm is reviewed, for additional details the reader is referred to [13]. In this paper the CUDA Toolkit 4.0 is used. As mentioned above using the CUDA technology we gain the possibility to exploit the massive compute power of modern GPUs by writing kernel programs e.g. in an extended version of C and execute these programs N times in parallel on the available hardware[14]. The program is executed by N different threads while no assumption can be made in which order the threads are executed. Started from the CPU a kernel is attached to a compute grid, which is separated into a number of blocks. In each block a specific amount of threads is running (see Figure 1).

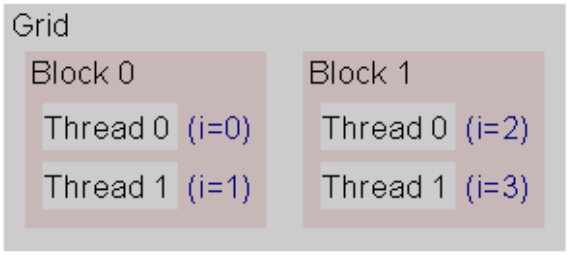

Figure 1: CUDA-programming model

C. Related Work

The potential of GPUs for parallel sparse matrix computations was recognized early on when GPU programming required shader languages. After the advent of NVIDIA CUDA, GPUs have drawn much more attention for sparse linear algebra and the solution of sparse linear systems. Existing implementations of the GPU Sparse matrix-vector product (SpMV) include CUDPP (CUDA Data Parallel Primitives) library [15], NVIDIA CUSPARSE library and the IBM SpMV library . In [16], different formats of sparse matrices are studied for producing high performance SpMV kernels on GPUs.

A GPU-accelerated Jacobi preconditioned CG method is studied in [17].The CG method with incomplete Poisson preconditioning is proposed for the Poisson problem on a multi-GPU platform.The Block-ILU preconditioned GMRES method is studied for solving sparse linear systems on the NVIDIA Tesla GPUs in. SpMV kernels on GPUs and the Block-ILU preconditioned GMRES method are used in for solving large sparse linear systems in black-oil and compositional flow simulations. A GPU implementation of deflated PCG method for bubbly flow computations is discussed in [18].

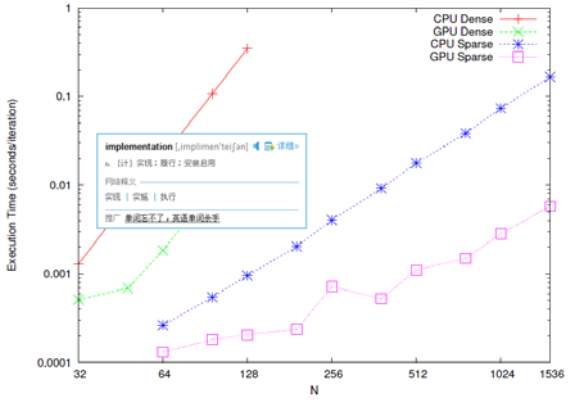

Figure 2: CUDA-execution Time

IV. Parallelized Preconditioned BiCGSTAB in CUDA

A.Sparse Matrix-Vector Product

In every iteration of the algorithm we have to compute several additions, dot products and a matrix-vector multiplication[19].

The threads of a block are executed as packages of 16 threads which is also called halfwarp. The threads inside a halfwarp are generally running in parallel. The blocks of a compute grid are executed sequentially but in case of available hardware resources blocks are dispatched and executed in parallel to other blocks. Every thread has two information: first in which block it is running and secound by which block internal number it is identified[20]. Hence the numbering of the threads can be done by,as in (2).

int i = blockIdx.x * blockDim.x + threadId.x.(2)

This is also illustrated in Figure 1. The next example demonstrates the addition of two vectors implemented in C for CUDA (see Algorithm 3).

```
global void VecAdd(float* A, float* B, float* C, int n)
    {
        int i = blockIdx.x * blockDim.x + threadId.x;
        if (i < n) C[i] = A[i] + B[i]
    }
```

Algorithm 3: CUDA-vector addition

The sparse matrix-vector product (SpMV) is one of the major components of any sparse matrixcomputations. However, the SpMV kernel which accounts for a big part of the cost of sparse iterative linear solvers, has difficulty in reaching a significant percentage of peak floating point throughput. It yields only a small fraction of the machine peak performance due to its indirect and irregular memory accesses[21]. Recently, several research groups have reported their implementations on high-performance sparse matrix-vector multiplication kernels on GPUs .NVIDIA researchers have demonstrated that their SpMV kernels can reach 15 GFLOPS and 10GFLOPS in single/double precision for unstructured mesh matrices. This section, will discuss implementation of SpMV GPU kernels using several different formats.

For a matrix-vector product, y = Ax, data reuse can be applied to the vector x which is readonly.Therefore, a good strategy is to place the vector x in the cached texture memory and access it by texture fetching . As reported in , caching improves the throughput by an average of 30% and 25% in single and double precision respectively[22].

B. SpMV in CSR Format

The compressed sparse row (CSR) format is probably the most widely used general-purpose sparse matrix storage format [23]. In the CSR format, the sparse matrix is specified by three arrays: one real/complex data array A which contains the non-zero elements row by row;To parallelize the SpMV for a matrix in the CSR format, a simple scheme called scalar CSR kernel is to assign each row to one thread. The computations of all threads are independent.

A better approach, termed vector CSR kernel, is to assign a half-warp (16 threads) instead of only one thread to each row. In this approach, the first half-warp works on row one, the second works on row two, and so on. Since all threads within a half-warp access non-zero elements of some row, these accesses are more likely to belong to the same memory segment. So the chance of

coalescing should be higher. But this approach incurs another problem related to computing vector dot products in parallel (dot product of a matrix row vector with the vector x). To solve this problem, each thread saves its partial result into the shared memory and a parallel reduction is used to sum up all partial results. Moreover, since a warp executes one common instruction at a time, threads within a warp are implicitly synchronized and so the synchronization in the half-warp parallel reduction can be omitted for better performance . In addition, full efficiency of the vector CSR kernel requires at least 16 non-zeros per row in A. Our vector CSR kernel is slightly different from the implementation in which a warp of threads are assigned to each row[24].

C. SpMV in ELL Format

The classical formats to store a sparse matrix (coordinate storage, compressed column storage, compressed row storage) are not applicable to parallelize the matrix vector product . In an efficient storage for sparse matrices the ELLPACK format was suggested, which was extended to the ELLPACK-R format especially for the use on GPUs. A matrix $A \in R^{n\times m}$ is represented by $Nz \in N$ the maximal number of elements unequal to zero per row,a representation matrix $A \in R^{n\times Nz}$ for the element unequal to zero,a representation matrix $j \in N_0^{n\times Nz}$ for the indice of the elements and an information vector $r1 \in N_0^n$ containing the number of the element per row.For

$$A=\begin{pmatrix}1&3&0\\0&1&1\\4&0&0\\0&0&2\end{pmatrix}\in R^{4\times 3}$$

with $Nz=2$ *we have the following ELLPACK−R representation*:

$$A=\begin{pmatrix}1&3\\1&1\\4&*\\2&*\end{pmatrix}\in R^{4\times 2}, j=\begin{pmatrix}0&1\\1&2\\0&*\\2&*\end{pmatrix}\in N_0^{4\times 2}, r1=\begin{pmatrix}2\\2\\1\\1\end{pmatrix}\in N_0^4$$

Let A be a matrix in ELLPACK-R representation. We realize the matrix-vector multiplication with n threads as seen in Algorithm 4[25].

```
Input : A ∈ R^{n×m}, v ∈ R^m
Output : u = Av ∈ R^n
for threadIndex ← x ← 0 to n − 1 do in parallel
  svalue ← 0
  max ← r1[x]
  for i ← 0 to max − 1 do
    value ← A[x + in]
    col ← j[x + in]
    svalue ← svalue + value · v[col]
  u[x] ← svalue
```

Algorithm 4: Matrix-vector multiplication in ELLPACK-R format.

The ELL formats are well-suited to matrices obtained from structured grids and semi-structured meshes[26]. CSR is a popular, general-purpose sparse matrix format which is convenient for many other sparse matrixcomputations. CSR kernels are subject to execution divergence caused by variations in the distribution of nonzeros per row.The COO SpMV kernel based on segmented reduction is robust with respect to variations in row sizes and others consistent performance. However, the COO format is verbose and has the worst computational intensity of the formats considered. Furthermore, the segmented reduction operation more expensive than alternative techniques that rely on a simpler decomposition of work into threads of execution. Nevertheless,the COO kernel is reliable and complements the deciencies of the other SpMV kernels（Figure 2）.

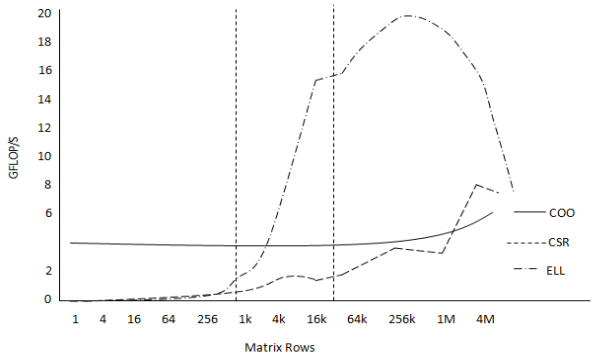

Figure 3: Exposing Parallelism

A direct implementation of SPAI on GPUs, however, is hampered by the observation that the cardinality of the index sets I_k and J_k cannot be obtained a-priori. Since dynamic memory allocation on GPUs using OpenCL is not possible by now, expensive scans for the required memory would be necessary. Therefore, the index sets I_k and J_k are set up on the CPU and then copied to GPU. After that, the matrices $A(I_k,J_k)$ are set up.The massively parallel architecture of GPUs suggests to solve all Least Squares problems simultaneously.However,overall memory requirements induced by the Least Squares problems may soon exceed the limited GPU RAM, thus only a subset can be processed at the same time. In our implementation the overall memory requirements on the GPU are computed from the size of the index sets I_k and J_k. If GPU RAM turns out to be too small, the work load is split into two chunks of equal size, which are processed one after another on the GPU. This strategy is then applied in a recursive manner to each of the two chunks.

Due to the high degree of parallelism still present in eachof the chunks, the full Least Squares problem including the QR factorizations of the blocks $A(I_k,J_k)$ is carried out on the GPU. For the better utilization of the SIMD architecture of GPUs[27], one thread per column of each block is used, so that each work group factors one SPAI block. After that, results are copied back to CPU RAM and the residual vector is computed.If the Euclidean norm of the residual is higher than a prescribed tolerance, sparsity pattern updates are carried out as described in the previous section.To summarize, our implementation of SPAI for a given initial sparsity

pattern is as follows[28]:
1. Determine the index sets I_k and J_k for each $k = 1, \ldots, n$.
2. Compute the memory consumption of the n Least Squares problems and split the work load into chunks.
3. For each chunk, assemble the matrices A_k and compute the solution of the Least Squares problems using QR factorization.
4. Copy the results back to CPU RAM and compute the residuals.
5. If further pattern updates are required, compute the augmented index sets I and J , otherwise go to 8.
6.Assemble the new entries in the augmented Least Squares matrix and compute the solution reusing previous QR factorizations.
7. Go back to 4.
8. Write all entries computed in the chunk to M.

D. Shared Memory and Texture Cache

In this section, To reduce the processing time the coefficient matrix was stored in ELLPACK-R format and the matrix vector operations were parallelized.Another approach to advance stability and acceleration is the preconditioning of the matrix. A sequential implementation of the preconditioned Bi-Conjugate Gradient Stabilized algorithm was implemented to evaluate this.Furthermore, the use of Shared Memory and Texture Cache could accelerate the runtime on older graphics cards[29].

In the implementation, which is described in this paper the input data were copied into the global memory of the graphics card. During the execution, the threads get the required data only from this memory, which is about 512 to 2048 MB on modern graphics hardware[30]. There exists also a comparatively small shared memory (it can also be configured as a L1-cache) and texture cache, which is only about a few KB. The access time to this kind of memory is much shorter (sometimes about a factor in the dimensions of 100) in comparison to the often uncached global memory. Because of the small size in the most cases it is not possible to store the whole coefficient matrix inside of it[31]. Another way to speed up the process is the skillful use of this memory. Therefore, it is necessary to copy parts of data for future calculations from global memory to shared memory and texture cache. In order to achieve an acceleration, this must be constructed so that most of these transfers occur in parallel to the running threads.

IV.EXPERIMENTS

This section provides a comparison of the runtimes of the sequential and the parallel implementation of the preconditioned BICGstab algorithm. As an application the possion equation is solved and different sizes of the n x n-coefficient matrix are chose. When solving a linear system the convergence speed of the conjugate gradient algorithm to compute an acceptably accurate solution strongly depends on the condition number of the system matrix. Therefore,the time per iteration is used for detailed comparison. The runtimes of the presented GPU version in CUDA 4.0.

In order to measure the GPU computation ability, we have used a Intel i7-3770 with 4 GB memory, hosting a NVIDIA Geforce 610M with 2 GB memory. All results using double-precision floating-point data. In most compute results,it is a good precision under 1e-10.All those matrices are available on the Sparse Matrix Florida Collection. Their properties are summarized in Table 1.

TABLE 1
DESCRIPTION OF THE MATRICES USED

Matrix	rows	cols	nonzero	description
poisson3Da	13,514	13,514	352,762	nonsymmetric
poisson3Db	85,623	85,623	2,374,949	nonsymmetric

Figure. 4 Each matrix illustrate the location of the nonzero elements. Some matrices are in close proximity to the diagonal, some elements are far from the diagonal.

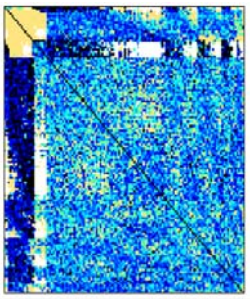

Figure 4 poisson3Da

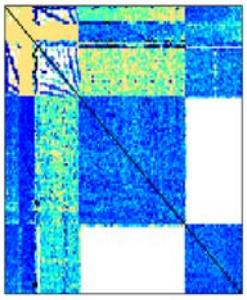

Figure 5 poisson3Db

In Table 2, We minimizes the execution time:ELL format for GPU and CSR format for CPU. The table contains the number of iterations required to reach the threshold condition.

TABLE 2
COMPARISON BETWEEN CPU AND GPU, TIMES IN S

Matrix	CPU times	GPU times	iter.
poisson3Da	6.54s	0.68s	81
poisson3Db	167.63 s	20.63 s	288

In Table 3, we inform the amount of floating point computing per second (CPU and GPU). As shown in the fourth column (size, number of nonzero elements,etc.). Solving a linear system in GPU is obviously less efficient than simply computing the SpMV. many operations are not really efficient on GPU.

TABLE 3
COMPARISON BETWEEN CPU AND GPU IN GFLOPS

Matrix	CPU Gflops	GPU Gflops	% of SpMV in solving
poisson3Da	0.68	6.11	56
poisson3Db	0.66	7.28	61

V.CONCLUSIONS

This work presents a preconditioned Bi-Conjugate Gradient Stabilized algorithm using the NVIDIA CUDA architecture.The operations which were parallelized are the addition, the dot product and the sparse matrix-vector multiplication. As sparse matrix format ELLPACK-R was used which provides an memory efficient storage for large coefficient matrices on the GPU. To compare the presented implementation, a comparison with a sequential

CPU implementation using the LAPACK based Armadillo C++ Library 1.0.0 and different CPU implementations was made. In comparisonto the sequential CPU implementations the parallel version of the Bi-Conjugate Gradient Stabilized algorithm is in average 8 to 10 times faster.

ACKNOWLEDGMENT

This work has been supported by National High-tech R&D Program (2009AA06Z108).

REFERENCES

[1] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser.A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform. Parallel, Distributed, and Network-Based Processing,Euromicro Conference on, 0:583–592, 2010.

[2] Michele Benzi.Preconditioning Techniques for Large Linear Systems: A Survey. Journal of Computational Physics 182, 418–477 (2002)

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

[4] Gu Tong-xiang Associate Professor, Doctor, Chi Xue-bin, Liu Xing-ping, Ainv and bilum preconditioning techniques, Applied Mathematics and Mechanics September 2004, Volume 25, Issue 9, pp 1012-1021

[5] J. Georgii and R. Westermann. A multigrid framework for real-time simulation of deformable volumes.In Proceedings of the 2nd Workshop On Virtual Reality Interaction and Physical Simulation,pages 507, 2005.

[6] J. R. GILBERT, Predicting structure in sparse matrix computations, SIAM J. Matrix Anal.Appl., 15 (1994), pp. 62-79.

[7] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards, 49:409,436, 1952.

[8] Y. Notay, Using approximate inverses in algebraic multilevel methods, *Numer. Math.* 80, 397 (1998).

[9] Y. Notay, A multilevel block incomplete factorization preconditioning, *Appl. Numer. Math*. 31, 209 (1999).

[10] NVIDIA-Corporation. Nvidia cuda c programming guide, Version 3.1, 2010.

[11] Y. Saad. Iterative methods for sparse linear systems.Second edition, 2003.

[12] L. N. Trefethen and D. Bau. Numerical Linear Algebra.SIAM: Society for Industrial and Applied Mathematics,1997.

[13] F. Vazquez, E. M. Garzon, J. Martinez, and J. J. Fernandez.The sparse matrix vector product on gpus.aceuales, pages 103, 2009.

[14] R. Freund, A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems, SIAM J.Sci. Comput. 14, 470 (1993).

[15] M. Benzi and M. Tuma, A sparse approximate inverse preconditioner for nonsymmetric linear systems, SIAM J. Sci. Comput. 19, 968 (1998).

[16] M. Benzi and M. Tuma, Numerical experiments with two approximate inverse preconditioners, BIT 38, 234(1998).

[17] M. Benzi and M. Tuma, A comparative study of sparse approximate inverse preconditioners, Appl. Numer. Math. 30, 305 (1999).

[18] L.Bergamaschi,G.Pini,andF. Sartoretto, Approximate inverse preconditioning in the parallel solution of sparse eigenproblems, Numer. Linear Algebra Appl. 7, 99 (2000).

[19] George Karypis and Vipin Kumar, Metis – a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version4.0, Tech. report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1998.

[20] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, Scan primitives for gpu computing, Graphics Hardware 2007, ACM, August 2007, pp. 97–106.

[21] Nathan Bell and Michael Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (New York, NY, USA), ACM, 2009, pp. 1–11.

[22] Serban Georgescu and Hiroshi Okuda, Conjugate gradients on graphic hardware: Performance& Feasibility, 2007.

[23] Rohit Gupta, A GPU implementation of a bubbly flow solver , Master's thesis, Delft Institute of Applied Mathematics, Delft University of Technology, 2628 BL Delft, The Netherlands,2009.

[24] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, Scan primitives for gpu computing, Graphics Hardware 2007, ACM, August 2007, pp. 97-106.

[25] Bell N, Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. NVIDIA technical report NVR-2008-004, NVIDIA Corporation, December 2008

[26] Vasily Volkov and James Demmel, LU, QR and Cholesky factorizations using vector capabilities of GPUs, Tech. report, Computer Science Division University ofCalifornia at Berkeley, 2008.

[27] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan, Solving sparse linear systems on nvidia tesla gpus, ICCS'09: Proceedings of the 9th International Conference on Computational Science (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 864–873.

[28] Marco Ament, Gunter Knittel, Daniel Weiskopf, Wolfgang Strasser, "A Parallel Preconditioned

Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform,"pdp, pp.583-592, 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010

[29] L. Buatois, G. Caumon, and B. Levy, “Concurrent number cruncher: a GPU implementation of a general sparse linear solver,” Int. J. Parallel Emerg. Distrib. Syst., 24(3):205–223,2009. ISSN 1744-5760

[30] Nvidia, “CUDA Toolkit 4.0 CUBLAS Library” NVIDIACorporation, Santa Clara, April, 2011

[31] Nvidia,”CUDA Programming Guide 4.0” NVIDIA Corporation, Santa Clara, April, 2011