GPU Accelerated Krylov Subspace Methods for Computational Electromagnetics

Sanjay Velamparambil#1, Sarah MacKinnon-Cormier, James Perry, Robson Lemos, Michal Okoniewski#2 Joshua Leon#3

Acceleware Corporation
1600 37th St. SW, Calgary, Alberta-T3E 3P1, Canada
#2 *Dept. Elect. & Comp. Engg., University of Calgary, Calgary, Canada*
#3 *Faculty of Engineering, Dalhousie University, Halifax, Canada*
#1 `sanjay.velamparambil@acceleware.com`
#2 `michal.okoniewski@acceleware.com`
#3 `Joshua.Leon@dal.ca`

Abstract— **Programmable graphics processor units (GPU), such as the NVIDIA^R Geforce 8800 series, offer a raw computing power that is often an order of magnitude larger than even the most modern multicore CPUs, making them a relatively inexpensive platform for high performance computing. In this paper, we report the development of two Krylov subspace solvers, the generalized minimal residual (GMRES) and the quasi-minimal residual (QMR) algorithms, on the GPU using the NVIDIA CUDA^R programming model[1]. The algorithms have been implemented as a stand-alone library. We report a speed-up of up to 13 times, on a single GPU, in our preliminary experiments with the classic problem of computing the capacitance of conductors using an integral equation method.**

I. Introduction

The demands of the computer gaming industry have pushed the development of graphics processing units (GPUs) with unprecedented raw power for numerical processing. GPUs such as the NVIDIA^R Geforce 8800 series have been demonstrated to have an order of magnitude larger processing power than the most modern multicore CPUs[1, Chapter 1]. However, until recently, harnessing this computing power for scientific computing purposes was possible only through OpenGL^R and DirectX^R commands[2], [3]. The recent introduction of NVIDIA's *compute unified device architecture* (CUDA) and and an application programming interface (API) using an extension to the C programming language, fortunately, has brought the GPU computing power closer to the computational scientist.

Numerical solution of electromagnetic problems most often lead to large, unsymmetric and indefinite matrix equations. Differential equation methods such as the finite element method (FEM) often lead to large sparse systems whereas integral equation methods generally lead to dense systems. Krylov subspace methods such as the GMRES and QMR are among the most frequently used workhorses used for solving these equations.

At Acceleware, we have implemented the GMRES and the QMR on NVIDIA GPUs using CUDA. The algorithms are implemented as a standalone library that can be easily plugged into existing solvers. In this paper, we report some preliminary performance results from our implementation.

The rest of the paper is organized as follows: in the next section, we summarize the salient characteristics of GPUs and the CUDA programming model. In section III, we outline the basic GMRES and QMR algorithms and their implementation on the GPU. In section IV, we present some numerical results from a classic electromagnetic problem, which is followed by conclusions in section V.

II. Computational Characteristics of GPUs and CUDA Programming Model

The superior computing power of the GPUs is a result of the fact that more transistors are devoted to data processing rather than data caching and flow control[1]. As a result, GPUs are well suited for data parallel operations with high arithmetic intensity. Until recently, however, accessing this computational power was possible only through graphics API, which was difficult and is inadequate for non-graphics application. Furthermore, GPUs supported only limited write access to on-board memory[1].

The CUDA programming model is a new hardware and software architecture that gives a simplified view of the GPU and that also allows much broader program control. As an API CUDA offers a minimal set of extensions to standard C language to facilitate GPU programming. For more details on CUDA, we refer the reader to the CUDA Programming Guide[1].

In the CUDA programming model, the GPU is viewed as a massively parallel, single instruction, multiple data (SIMD) compute device. For example, the NVIDIA G8800 GTX processor consists of 16 *multiprocessors*, with each multiprocessor consisting of 8 processors. Computations are carried out by running large numbers of light weight *threads* executing identical program segment, called a *kernel*, operating on different sets of data (SIMD paradigm).

Some of the limitations of GPU computing are:

- SIMD parallelism discourages branching in programs. Thus, it is imperative that the kernels must have as little indirections and branching as possible.

- As of now, only one kernel can be executed at any time on the GPU.
- The data transfer rate within the GPU is much higher that between the device memory and the host memory. Thus, the performance is often dictated by the rate at which data can be transferred between the GPU and the host memory.
- As of the time of writing, only single precision floating point computations are supported in the GPU.

The process of developing an efficient algorithm and the flow of computation can thus be summarized as:

- Identify the data parallel computations in the code. Abstract each such computation into simpler functions, called kernels.
- Download the data to be operated on to the GPU.
- Download the kernel and execute on the data.
- Bring the results back to the host memory.

In developing the Krylov subspace solvers described below, we have chosen a hybrid approach to GPU based computing. We perform all the arithmetic intensive computations on the GPU whereas the more logically complex parts, those which involve branching and decision making, on the CPU. This gives rise to the superior performance discussed in section IV.

III. Krylov Subspace Methods on the GPU: GMRES and QMR

Starting from an initial guess x_0, Krylov subspace methods iteratively construct an approximate solution to the equation

$$Ax = b; \quad A \in \mathbb{R}^{n\times n}; x, b \in \mathbb{R}^n \tag{1}$$

from the space $\mathcal{K}_m(r_0, A) = \text{span}(r_0, Ar_0, \cdots, A^{m-1}r_0)$[4]. Without the loss of generality, we shall assume that these the matrices are real and nonsymmetric. Extension to complex, non-Hermitian matrices is, in general, straight forward.

The computationally most intensive parts in each iteration essentially consist of one or more of the following algebraic operations

- For given $z \in \mathbb{R}^n$, compute $w = Az$ or $w = A^Tz$ or both.
- For given $x_1, x_2, \cdots, x_p \in \mathbb{R}^n$ and given $\alpha_i \in \mathbb{R}$, form $y = \sum_{i=1}^{p} \alpha_i x_i$.
- For given $x \in \mathbb{R}^n$, compute $\|x\|_2$.
- For given $x, y \in \mathbb{R}^n$, compute y^Tx.

We have developed kernels for performing these operations, especially Az and A^Tz when A is sparse, efficiently on the GPU. All our Krylov subspace solvers are built on top of this library.

A. GMRES

Our implementation of the GMRES algorithm closely follows Algorithm 6.9 in [4]. Starting with the initial residual $r_0 = b - Ax_0$, this algorithm constructs a basis for the Krylov subspace $\mathcal{K}_m(r_0, A)$ using Arnoldi process with modified Gram-Schmidt (MGS) orthogonalization. Please refer to [4, Chapter 6] for more details.

In our implementation, the matrix A, the right hand side vector b and the initial guess x_0 are first downloaded to the GPU. The MGS-Arnoldi process is then performed as a hybrid CPU-GPU kernel, leading to the construction of the Hessenberg matrix H_m [4, Algorithm 6.9] on the CPU. Note that the basis vectors V_m to $\mathcal{K}_m(r_0, A)$ are generated and stored in the GPU. Solution to the least squares problem is then computed through the standard processes of Given's rotations and back substitution on the CPU[4].

This approach allows us to do the computationally insignificant operations, such as the Given's rotations, and logical decision making, such as convergence checks and restarts, on the CPU. Furthermore, although the MGS-Arnoldi step is inherently sequential, it does not affect the performance noticeably.

B. QMR

The essential difference between GMRES and QMR is that in QMR, the Arnoldi process is replaced with Lanczos bi-orthogonalization procedure, leading to the construction of bases for $\mathcal{K}_m(r_0, A)$ and $\mathcal{K}_m(r_0, A^T)$. Our algorithm implements a variation of the QMR described in [5], [6] without the look-ahead step. We refer the reader to [5], [6] for more detailed discussion of the algorithm.

One important consequence of the Lanczos bi-orthogonalization process is that it leads to a three term recurrence relation between basis vectors for $\mathcal{K}_m(r_0, A)$ and $\mathcal{K}_m(r_0, A^T)$. As a result, the storage requirements for QMR, in general, are considerably lower than that of GMRES. However, QMR requires both Ax and A^Tx in each iteration, which can be computationally expensive.

As in the case of GMRES, we download the matrix A and the vectors x, b in to the GPU at the start of computations. The Lanczos process is run on the GPU and the basis vectors generated are kept in the GPU as well. The three term recurrence in the Lanczos procedure is computed through a highly optimized kernel function. However, as in the case of GMRES, the least square solution to the minimization problem, through Given's rotations, is performed in the CPU. The solution vector is updated on the GPU through the same kernel call used in the Lanczos process.

IV. Preliminary Numerical Results

In order to demonstrate the speed-up that can be achieved through our GPU based Krylov subspace solvers, we consider a classic problem in electromagnetics: computation of capacitance of a set of conductors using an integral equation method. A note about the choice of the problem is in order. As an independent software vendor, Acceleware Corporation does not develop full-fledged electromagnetic solvers. At the time of writing this paper, we do not have access to solvers that can generate more realistic examples for illustration. As a result, we chose to implement a classical example that would demonstrate our solvers without stretching our resources.

We consider the problem of computing the capacitance of a perfect conductor situated in free space. The conductor is charged to a potential of ϕ_0V. Let the charge density induced on the surface S be denoted by σ. Then the capacitance is defined as $C = \int_S \sigma \, dS/\phi_0$.

To compute the charge density, we need to solve the integral equation

$$\int_S \frac{\sigma(\vec{r}')}{|\vec{r} - \vec{r}'|} dS' = 4\pi\epsilon_0\phi_0. \tag{2}$$

Towards this end, we approximate the surface S by a number of triangles $\Delta_i, 1 \leq i \leq n$, and the unknown surface charge density σ using the pulse basis functions[7]. The matrix equation is generated using Galerkin testing, leading to the matrix equation

$$Ax = b \tag{3}$$

$$\text{where } x_i = \sigma_i \tag{4}$$

$$b_i = |\Delta_i|\phi_0 \tag{5}$$

$$A_{ij} = \int_{\Delta_i}\int_{\Delta_j} \frac{1}{|\vec{r} - \vec{r}'|} dS' dS, \quad \text{for } 1 \leq i, j \leq n \tag{6}$$

Note that this is a dense matrix equation.

For our numerical examples, we consider a unit sphere charged to 1V. The exact value of capacitance is $1.11126 \times 10^{-10}F$. We solved the problem for matrix sizes ranging from 4000 to about 16,000 triangles. Obviously, this is an overkill from the point of view of the specific electromagnetic problem at hand. However, it serves our purpose of demonstrating the speed-up in the solution time.

All the computations were performed on a workstation with two dual-core AMD Opteron 2.8GHz processors, and having 16GB of RAM, although the CPU computations used only one processor. This is because our CPU implementations used Intel^R MKL performance libraries and the Level 1 and Level 2 BLAS operations of MKL, which are necessary for GMRES/QMR are *not* multithreaded. The graphics processor used was NVIDIA Quadro FX 5600 with 1.5GB of RAM on the device. In all cases, the number of iterations required was about 7, both on CPU and the GPU, indicating a well conditioned system. Moreover, the capacitance values computed on both devices matched to four decimal digits when compared to the analytical value.

TABLE I

COMPARISON OF THE CPU AND GPU VERSIONS OF GMRES. N IS THE NUMBER OF TRIANGLES (UNKNOWNS)

N	CPU time(s)	GPU time (s)	Speed-up
4608	0.391	3.1E-2	12.6
6272	0.5470	6.3E-2	8.6
8192	1.234	9.4E-2	13.13
12,800	1.906	0.234	8.14
15,488	2.765	0.344	8.03

In Table I, we present the time for solving the matrix equation using GMRES. Note that we solving the equation for one right hand side. It is clear that the speed-up is impressive. The superior speed-up for $N = 4608$ and $N = 8192$ is due to the fact that the computation of Ax for these sizes is more optimal. We are currently working on better algorithms to even out the performance.

However, an important question arises: the actual time for each of these computations is very small, the maximum being about 3 seconds on the CPU. *So is it worth accelerating such computations?* To answer this question, we note that in this case we are solving a very well conditioned system for one right hand side. However, in most practical cases, there will many, often dozens of, right hand side vectors. Furthermore, the systems need not be well conditioned, requiring far more number of iterations than what is reported here. In such situations, the cumulative savings will be far more impressive than what this preliminary result shows.

A note sparse systems is called for. Our implementations of the algorithms are agnostic to the type of matrices. We have also developed highly efficient kernels for handling Ax for sparse matrices. The performance of our solvers in some initial testing with a few publicly available, large, sparse systems, involving 10^5 or more unknowns was at least equally good or superior to those reported here.

V. CONCLUSIONS

In this paper, we have reported the implementation of Krylov subspace methods on graphics processing units. We have used the NVIDIA CUDA programming model for our implementations. We have implemented the GMRES and QMR algorithms on the GPU. For efficiency, we have isolated the compute intensive operations and performed them on the GPU whereas the logic intensive operations were done on the CPU, resulting in a practical, hybrid paradigm. We have demonstrated the performance of the GMRES algorithm for a classic electrostatic problem and we have shown that significant speed-up can be achieved.

Work is currently under progress to develop complex versions of these algorithms. We are also in the process of extensively testing the performance for various classes of matrices, from a variety of physical problems. Another ongoing work is the porting of the implementation to multiple GPUs. The progress in these directions will be reported at later date.

REFERENCES

[1] NVIDIA. (2007, Nov.) NVIDIA CUDA programming guide. NVIDIA_CUDA_Programming_Guide_1.1.pdf. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/-NVIDIA_CUDA_Programming_Guide_1.1.pdf

[2] OpenGL^R. (2007, Nov.). [Online]. Available: http://www.opengl.org

[3] Microsoft Corporation. (2007, Nov.) Directx developer center. [Online]. Available: http://msdn2.microsoft.com/en-us/directx/default.aspx

[4] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[5] R. W. Freund, M. H. Gutknecht, and N. M. Nachtigal, "An implementation of the look-ahead lanczos algorithm for non-hermitian matrices," *SIAM J. Sci. Comput.*, vol. 14, no. 1, pp. 137–158, Jan. 1993.

[6] R. W. Freund and N. M. Nachtigal, "QMR: a qusi-minimal residual method for non-hermitian linear systems," *Numer. Math*, vol. 60, pp. 315–339, 1991.

[7] S. Rao, A. Glisson, D. Wilton, and B. Vidula, "A simple numerical solution procedure for statics problems involving arbitrary-shaped surfaces," *Antennas and Propagation, IEEE Transactions on [legacy, pre - 1988]*, vol. 27, no. 5, pp. 604–608, Sep 1979.