Accelerating the GMRES Iterative Linear Solver of an Oil Reservoir Simulator using the Multi-Processing Power of Compute Unified Device Architecture of Graphics Cards

Nima Ghaemian[1], Asaad Abdollahzadeh[2], Zoltan Heinemann[3], Hossein Beyrami[2], Andreas Harrer[4], Mohsen Sharifi[1], and Gabor Heinemann[4]

[1] Iran University of Science and Technology, Iran
[2] Research Institute of Petroleum Industry of Iran[‡], Iran
[3] Mining University of Leoben, Austria
[4] Heinemann Oil GmbH, Austria

Abstract. The Generalized Minimal Residual Method (GMRES) is an iterative method for numerical solution of a system of linear equations. It approximates the solution by a vector in a Krylov subspace with minimal residual, using the Arnoldi iteration to find the vector. To gain higher performance, we have applied GMRES to the Solver of an in-house Reservoir Simulator Software that has been basically designed and operated as an ordinary CPU-based program. We have used the multi-processing capabilities of the Graphics Processor Unit (GPU) of a specific brand of a Graphics Card to our advantage, using Compute Unified Device Architecture (CUDA). As well as presenting our acceleration technique, we show the acceleration rate experimentally.

Key words: Reservoir Simulator, Iterative Numerical Method, GPU, CUDA, GMRES

1 Introduction

Solution methods for linear systems fall into two large categories: direct methods and iteration methods. Direct methods determine the solution to the linear system exactly up to machine precision. They perform in a robust manner leading to the solution after a finite number of floating-point operations. Nevertheless, the drawback of direct methods is that they are expensive in computation time and computer memory requirements and therefore cannot be applied to the solution of linear systems of very large order. The efficient solution of large systems generally requires the use of iteration methods which work by generating sequences of improving approximate solutions. One main category of iterative methods is preconditioned Krylov subspace methods.

[‡] The experiments were conducted with the support of Research Institute of Petroleum Industry of Iran.

2 The GMRES Method

A system of linear equations to be solved is denoted by $Ax = B$. The matrix A is assumed to be invertible of size m-by-m. The nth Krylov subspace for this problem is:

$$K_n = \mathsf{span}\{b, Ab, A^2b, ..., A^{n-1}b\}. \tag{1}$$

GMRES approximates the exact solution of $Ax = B$ by the vector $x_n \in K_n$ that minimizes the norm of the residual $Ax_n - b$. The vectors $b, Ab, A^2b, ..., A^{n-1}b$ are almost linearly dependent, so instead of this basis, the Arnoldi iteration is used to find orthonormal vectors $q_1, q_2, ..., q_n$ which form a basis for K_n. Hence, the vector $x_n \in K_n$ can be written as $x_n = Q_n y_n$ with $y_n \in R^n$, where Q_n is the m-by-n matrix formed by $q_1, ..., q_n$.

The Arnoldi process also produces an $(n+1)$-by-n upper Hessenberg matrix $\bar{H}_n$ with $AQ_n = AQ_{n+1}\bar{H}_n$ Because Q_n is orthogonal, we have

$$\|Ax_n - b\| = \|\bar{H}_n y_n - e_1\|, \tag{2}$$

Where $e_1 = (1, 0, 0, ..., 0)$ is the first vector in the standard basis of R^{n+1}. Hence, x_n can be found by minimizing the norm of the residual $r_n = \bar{H}_n y_n - e_1$. This is a linear least squares problem of size n. This yields the GMRES method. At every step of the iteration:

1. do one step of the Arnoldi method;
2. find the y_n which minimizes $\|r_n\|$;
3. compute $x_n = Q_n y_n$;
4. repeat if the residual is not yet small enough.

At every iteration, a matrix-vector product Aq_n must be computed. This costs about $2m^2$ floating-point operations for general dense matrices of size m, but the cost can decrease to $O(m)$ for sparse matrices. In addition to the matrix-vector product, $O(nm)$ floating-point operations must be computed in the nth iteration [1].

3 Compute Unified Device Architecture

In a matter of just a few years, the programmable graphics processor unit has evolved into an absolute computing workhorse. With multiple cores driven by very high memory bandwidth, today's GPUs offer incredible resources for both graphics and non-graphics processing [3].

CUDA is a technology of NVIDIA® and stands for Compute Unified Device Architecture and is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need to map them to a graphics API. The operating system's multitasking mechanism is responsible for managing access to the GPU by several CUDA and graphics applications running concurrently [3].

When programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or host. In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

CUDA has hierarchical and non-unified memory access and Both the host and the device maintain their own DRAM, referred to as host memory and device memory, respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's high-speed Direct Memory Access (DMA) engines [3].

4 Approach

As part of the development process of a reservoir simulator, namely PRS, acceleration of the GMRES solver is considered. The whole code of the simulator is written in FORTRAN 95 and any additional facility to the simulator should be ported to FORTRAN. Moreover, a bulk of dusty-deck code is available and reconstructing those codes in another language to support the state-of-the-art technology seems to be expensive.

In our case, the solver module of the simulator uses GMRES solver to solve equations. There are two ways to speed up the solver by CUDA: 1) keeping at least the spirit of the current algorithm, and 2) developing a totally new algorithm considering the new architecture. We chose the first alternative believing that the second one is quite timely.

To keep the current algorithm, we had two choices:

1. Handling GPU as a vector-machine and replacing the primitives that have vectorized alternatives in CUDA with the old primitives in the FORTRAN dusty-deck code.
2. Partially-modifying the algorithm without changing its underlying paradigm, implementing it in CUDA, and then porting it to FORTRAN

4.1 GPU, as a Vector Machine

We found two time-consuming and vectorizable primitives. Those primitives were determined as the Basic Linear Algebra System (BLAS) primitives. CUDA has provided the CUDA-enabled Basic Linear Algebra System (CUBLAS). There were two alternatives to port the CUBLAS to FORTRAN:

1. Thunking the CUBLAS
2. Using the non-thunked CUBLAS

Thunking CUBLAS means wrapping the details of CUBLAS and keeping the FORTRAN code unchanged. Using non-thunked CUBLAS means writing a lightweight wrapper for BLAS functions and modifying the Fortran Code to perform additional operations such as initialization/termination of CUBLAS, allocation/deallocation etc. The results show that the second alternative shows overwhelming

speed-up but one of the primitives falls down in Von-Neumann bottleneck, e.g. communication overhead between main memory and device memory, and reduces the overall speed.

4.2 Implementing the Algorithm in CUDA

Since there is considerable amount of iterations within the GMRES code, we opted to implement the code in CUDA multithreading environment. We separated the preconditioning module and focused on GMRES module.

To transform the dusty-deck FORTRAN code in CUDA, we developed a semi-automatic procedure. The phases of the procedure are as follows:

1. Transforming the available code to an intermediate code
2. Performing the *dependency analysis* and refining the code
3. Performing the *unstructured jump analysis* and re-structuring the code
4. Exploiting the hidden concurrency and labeling the code
5. Transforming the intermediate code to CUDA code
6. Optimizing the CUDA code considering architecture-specific concerns

The resulting CUDA code was run on a platform with the following specification:

- Intel® Pentium® D CPU, clock speed 2.66GHz and FSB speed 533MHz
- 1GB of RAM
- NVIDIA® Tesla C870 GPU, 1.5GB dedicated memory

The test scenarios run indicated a 60% speed up compared to the runs of the original solver.

5 Related Works

Although CUDA is a new architecture, there is a large amount of related work in the academic and a few in industrial scales. However, they have mostly not been applied to the field of equation solvers. Only [4] reports a practice on CG solver.

References

1. Y. Saad and M.H. Schultz: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput., 7:856-869, doi:10.1137/0907058. (1986)
2. J. Stoer and R. Bulirsch: Introduction to numerical analysis, 3rd edition, Springer, New York. ISBN 978-0-387-95452-3. (2002)
3. NVIDIA CUDA development team: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, NVIDIA corporation (2007)
4. W.A. Wiggers,V. Bakker,A.B.J. Kokkeler and G.J.M. Smit: Implementing the conjugate gradient algorithm on multi-core systems, IEEE International Symposium on System-on-Chip, Finland (2007)