Implementation and Performance Analysis of a Parallel Oil Reservoir Simulator Tool using a CG Method on a GPU-Based System

Leila Ismail

Computer and Software Engineering/ HPGCL Research Lab
College of IT
UAE University
Al-Ain, UAE
Email: leila@uaeu.ac.ae (Correspondence Author)

Jamal Abou-Kassem

Department of Chemical and Petroleum Engineering
College of Engineering
UAE University
Al-Ain, UAE
Email: j.aboukassem@uaeu.ac.ae

Bibrak Qamar

HPGCL Research Lab
College of IT
UAE University
Al-Ain, UAE

***Abstract*—An oil reservoir simulator is a crucial tool used by petroleum engineering to analyze reservoir conditions. To increase its performance, we implement a parallel version of the tool on a Graphic Processing Unit (GPU), using Computer Unified Device Architecture (CUDA) programming model and the Single Instruction Multiple Threads (SIMT). This paper presents our parallel implementation and performance analysis for 1-D, 2-D, and 3-D oil-phase reservoirs. The implementation and the performance evaluation reveal the gains and the losses achieved by the parallelization of a reservoir simulator on a Graphics Processing Unite (GPU) system. The performance results show that despite the interdependency between the different computational parts of the Conjugate Gradient (CG) method used as a linear solver in the parallel reservoirs, a speedup of 26 can be easily obtained for an oil reservoir simulator using 15 streaming multiprocessors (SMs), compared to a sequential CPU execution. The parallel execution scales well with grid dimensionality.**

Keywords–Oil Reservoir Simulator; Conjugate Gradient (CG) Method; GPU; High Performance Computing

I. INTRODUCTION

In order to correctly make decisions on how to recover hydrocarbons from a reservoir, an accurate numerical model of the reservoir must be established to enumerate the outcomes and performance under different operating conditions, such as, the location and the rate of the injection in wells, and the recovery techniques. Oil reservoir simulators [[1]-[7]] are then crucial to the oil production, as they help to avoid the high costs of drilling unnecessary wells.

In an oil reservoir simulator, the physical characteristics of the reservoir, such as rock properties, fluid properties, and fluid-rock properties, and physics form the basis of Partial Differential Equations (PDE) which define the reservoir characteristics and the physical processes occurring in it. The partial differential equations representing the reservoir characteristics are discretized and expressed over these cells using algebraic equations, giving rise to a large system of linear equations. The latter are solved using numerical solvers, the most consuming chunk of CPU time of the simulator. The CG method [20] is an iterative numerical solver to solve a system of N linear equations with N unknowns. CG is very attractive to the oil industry thanks to its small memory requirements. In addition, the original coefficient matrix does not change during CG computations which preserve the original matrix generated by the oil simulator. All these properties of the CG method lead us to chose the CG method as a solver to parallelize on the GPU system and to introduce in our parallel oil reservoir simulator [31], [32], [33].

Oil and gas industry starts exploring the GPU system to develop parallel oil reservoir simulators [35], [34]. The GPU [36] is a specialized circuit designed to rapidly manipulate memory, consisting of multiple streaming multiprocessors (SMs), each consisting of 32 cores, each of which can execute one floating point or integer instruction per clock. Initially designed for graphics processing, the GPU is a promising technology for parallel computing-intensive applications. In this work, we implement a sequential and a parallel version of the 1-D, 2-D, and 3-D oil reservoir simulators on a GPU system. The sequential implementations do not use any optimized or parallelized sparse linear algebra library.

Section II describes the overall implementation of the simulator. The parallelization relies on a parallel CG method that we designed and implemented for the reservoirs. In section III, we review related works on CG implementations and sparse matrix-vector multiplication, which is a key CG operation. The programming model of the CG including algorithms and sequential and parallel implementations are

978-1-4799-4922-9/14 $31.00 © 2014 IEEE
DOI 10.1109/UKSim.2014.113

described in section IV. Our experiments and analysis of results are presented in section V. Finally, VI concludes our work.

II. Overall Implementation of the Oil Reservoir Simulator

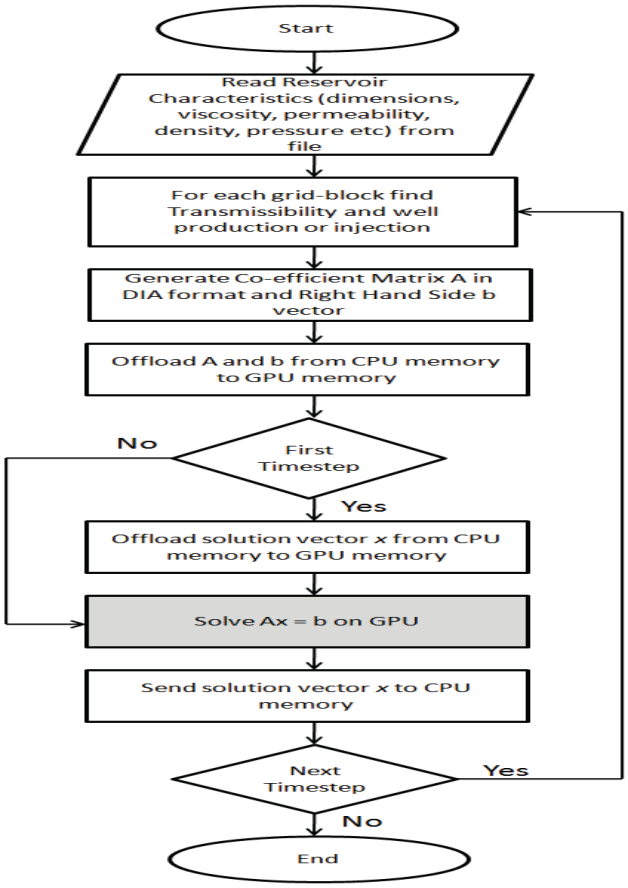

Figure 1. Oil Reservoir Simulator Algorithm.

Figure 1 shows the flow chart representing our implementation of the oil reservoir simulator.

The simulator first reads from an input data file the characteristics of the reservoir to be modeled. The characteristics constitute of the followings: the size of reservoir, the number of grid blocks and the dimensions of each grid block, the location of the wells and the production rates, the reservoir rock properties, the porosity, the permeability, the the reservoir fluid properties, the formation volume factor, and the viscosity.

The Transmissibility for each grid block is then computed using the following formula:

$$T_x = \beta_c \frac{A_x K_x}{\mu B \Delta_x} \quad (1)$$

The above formula is the transmissibility in the x-direction where μ is the viscosity, B is the formation volume factor, Δ_x is the dimension of a grid block in the x-direction, A_x is the area of a grid block (y-dimension * height), K_x is the permeability in the x-direction and β_c is the transmissibility conversion factor.

Based on the values computed for the transmissibility and the production rates of the wells, the simulator then generates the system $Ax = d$, where A is the coefficient matrix, x is the solution vector for the value of pressure in each grid block of the reservoir and d is is a known vector which represents the right hand side of the generated system of equations. The system $Ax = d$ is generated using the following formula for each grid block, at the time step $n + 1$:

$$\sum_{l \in \varphi_n} T_{ln}^{n+1} \left[(p_l^{n+1} - p_n^{n+1}) - \gamma_{l,n}^n (Z_l - Z_n) \right] + \sum_{l \in \epsilon_n} (q_{sc_{l,n}}^{n+1} + q_{sc_n}^{n+1}) = \frac{V_{b_n}}{\alpha_c \Delta_t} \left[\left(\left(\frac{\phi S}{B} \right)_n^{n+1} - \left(\frac{\phi S}{B} \right)_n^n \right) \right] \quad (2)$$

where T is the transmissibility, $q_{sc_{l,n}}$ is the flow from boundary block l to block n, q_{sc_n} is the well production rate at block n, γ is gravity and Z is the elevation, V is the bulk volume of the block n, S is the saturation and ϕ is the porosity, and Δ_t is the time step.

In our implementation, A and d are offloaded to the GPU memory and the solution vector x which contains the pressure at each grid block is offload only once to the GPU memory, since afterwards x is updated in GPU memory for consecutive simulator time steps. Once A, x and d are in the GPU memory, the simulator launches the solver (CG) at the GPU for the calculation of the pressure for the next time step at each grid block of the reservoir.

When the calculation of the new pressure for each grid block at the GPU level is completed, the vector x is transferred from the GPU memory to the CPU memory to be used to update the matrix A and the vector d, which are used for the next time step.

III. Related Works

A. Prior Implementations of Conjugate Gradient

Several algorithms have been published to parallelize CG [21], [22], [12], [15]. In references [21] and [22], algorithms have been implemented on a specialized event-driven multi-threaded platform. In references [12] and [15], algorithms have been implemented on a distributed shared memory cluster. Reference [23] introduces a parallel algorithm for matrix-vector multiplication, considers a particular matrix

with regular sparsity and studies the impact of mesh partitioning on the performance. References [26] and [27] introduce data decomposition strategies for matrix-vector multiplications to increase CG efficiency on hypercubes and meshes network for unstructured sparse matrices. Blocks of matrix are assigned to processors to perform partial result of the matrix-vector multiplication. Non-zeros are transferred using storage techniques of non-zeros in a vector. They concluded that the broadcast-based approach does not scale with increasing number of processors. By using an overlap mechanisms for global summation to hide communication cost involved in the parallel CG, a speedup of 2.5 was obtained on 128 processors compared to the original NAS parallel benchmark [19]. References [28] and [29] divide the overall CG algorithm into blocks of algorithms to reduce communication, but did not address the cost of communication among the blocks. Jordan et al. Reference [24] reported experiments where matrix-vector multiplication loads are distributed based on processors speed in a heterogeneous cluster, using a dense matrix. The parallel algorithm suggested by Kim et al [11] explored the parallel CG with both Jacobi diagonal preconditioning and incomplete Cholesky factorization preconditioning on a number of different meshfree analysis applications. They investigated the parallel performance of the solver using a homogeneous cluster and obtained a speedup of 12 compared to sequential execution in both cases, using 12 processors.

B. Sparse Matrix-Vector Multiplication

Sparse matrices arise in different problems of scientific and engineering nature. They can range from a very well structured to those which are highly irregular, where the non-zero values in different rows do not follow a distribution pattern. Methods have been developed on the ways to store the non-zero values of a sparse matrix to reduce memory storage requirements.

There are a large spectrum of sparse matrix representations, each with different storage requirements, computational characteristics, and memory-access method. Different storage formats have been introduced, and the DIA (diagonal format) was found to be suited for GPU implementations when the nonzero values are restricted to a small number of matrix diagonals [13], which is the case of the matrix generated by an oil reservoir simulator. However, the downsides of the DIA is that it allocates storage of values which are not part of the matrix and explicitly stores zero values in occupied diagonals. We have then to introduce an extra information to differentiate between a zero value which comes from a diagonal and a zero value which comes from the DIA mechanism. Coordinate format and compressed sparse row format, such as CSR (Compressed Storage Row), used for instance in reference [27], are general purpose format for storing the positions and the data of non-zero values in a matrix. Other formats are explored in reference [13]. In this work, we use DIA format for the GPU parallel implementation, as the coefficient matrices generated in 1-D, 2-D and 3-D oil reservoirs are diagonals-based structured matrices with few zeros in diagonals. However, we use the CSR format for the sequential CPU implementation, as it gives better performance on CPU than GPU [13].

In DIA, every matrix is represented by 2 arrays: an array which stores the non-zero values of the diagonals and an array which stores the offset of each diagonal from the main diagonal. By convention, the offset for main diagonal is 0, while the offset for k^{th} super-diagonal is k and the k^{th} sub-diagonal is -k. As the indices of rows and columns of nonzero entries are not stored explicitly, they are implicitly defined by their position within a diagonal and offset between diagonals. DIA format reduces the memory size for storing matrix and decreases the amount of memory transfers and provides coalesced memory access during sparse matrix-vector multiplication (SpMV) [13]. This makes DIA format appropriate in our case as GPUs have relatively less memory and perform poorly when memory is accessed irregularly [14].

IV. PROGRAMMING MODEL

A. CG Algorithm

As shown in Figure 5, the CG method starts with a random initial guess of the solution x_0 $(step1)$. Then, it proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution $(step10)$, residuals corresponding to the iterates $(step11)$, and search directions used in updating the iterates and residuals $(step14)$. Although the length of these sequences can become large, only a small number of vectors need to be kept in memory. In every iteration of the method, two inner products $(insteps9and13)$ are performed in order to compute update scalars $(steps9and13)$ that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric and positive definite linear system, these conditions imply that the distance to the true solution is minimized in some norm $(step12)$.

B. Matrix Structure

Figure 2 shows a computational mesh for a discretized 3-D oil reservoir, where N_x is the number of cells of the oil reservoir in the x direction, N_y is the number of cells of the reservoir in the y direction and N_z is the number of cells of the reservoir in the z direction. $N = N_x N_y N_z$ is the generated matrix size. The mesh reveals the way the unknown vector is composed. By numbering the unknowns, the outcoming linear system of equations following a discretization is represented by a heptadiagonal structured sparse matrix. A 2-D computational mesh is a simplified version of a 3-D computational mesh, where only N_x and N_y are considered to produce the matrix. A 2-D computational mesh produce a pentadiagonal structure.

In a 1-D computational mesh, only the N_x is considered producing a tridiagonal matrix structure.

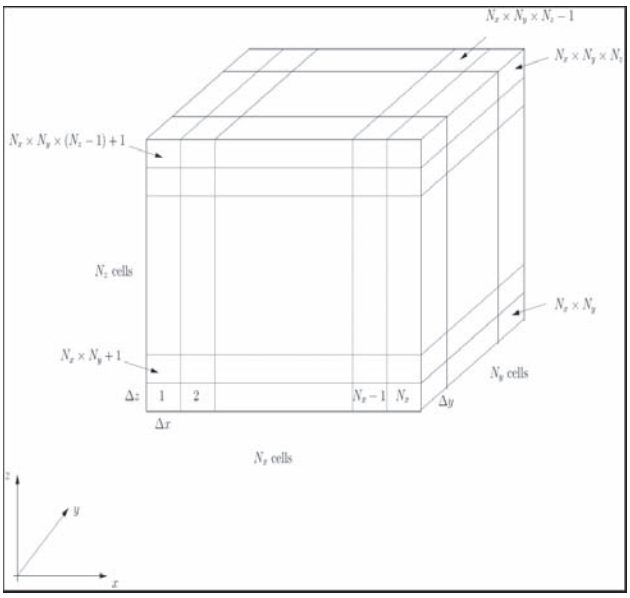

Figure 2. Computational mesh for discretized 3-D oil reservoir.

1) Parallel Implementation: The main goal of this work is to divide the CG operations by the number of available threads to increase the CG performance vis-a-vis its sequential execution. The algorithm presented in Figure 5 presents 2 types of divisible loads: 1) the sparse matrix-vector multiplication (SpMV) presented in step 8 of Figure 5, and 2) the scalar-vector and/or vector-vector operations presented in steps 9, 10, 11, 13, and 14. The divisible loads are dispatched to multiple threads. We use thread blocks because threads within a block can synchronize. To parallelize the matrix-vector multiplication, a matrix, expressed in DIA format, is divided into parts so that each thread can access the memory and work on its part of the matrix and perform matrix-vector multiplication in parallel to other threads. As shown in Figure 3, each tread is assigned to one row of the matrix and computes one element of the resultant vector, corresponding to that row. Since the matrix A is stored in DIA format, then accesses to the matrix A and the vector x are coalesced increasing bandwidth of global memory access. To parallelize the vector-vector multiplication, the sequential addressing is used and the threads can synchronize by storing their intermediate results which are then accumulated by the thread block which finishes last as shown in Figure 4.

Since global synchronization is not possible among threads from different thread blocks [37], we divided the CG algorithm into kernels as shown in Figure 5. The main CG loop and some scalar operations run on CPU, while compute intensive parts SpMV and vector-vector multiplication are offloaded to GPU.

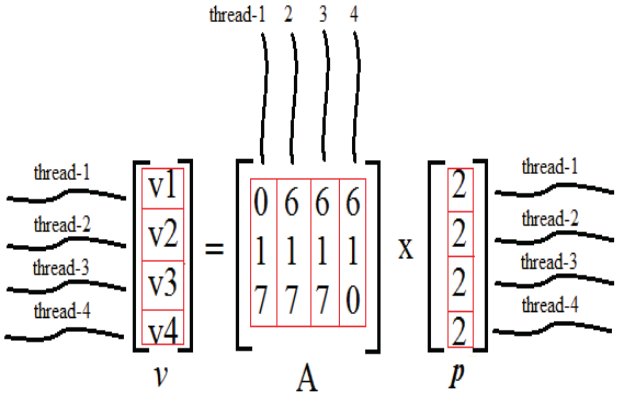

Figure 3. SpMV using DIA format: the matrix size is 4x4 and the total number of threads is 4.

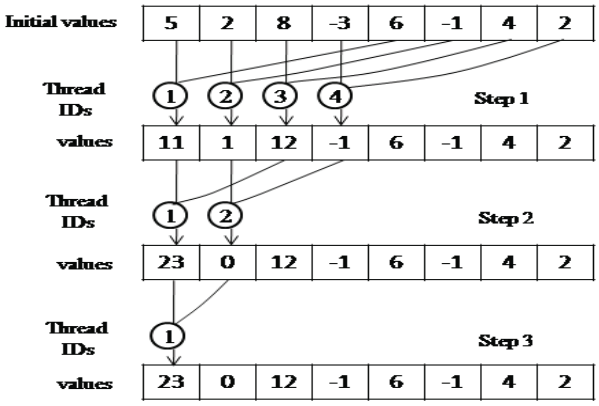

Figure 4. Reduction operation of vector-vector multiplication using sequential addressing.

V. PERFORMANCE ANALYSIS

In this section, we analyze the performance of our CG method parallel implementation on GPU using CUDA.

A. Experimental Environment

The experiments are conducted on a Nvidia GeForce GTX 480 GPU which consists of 15 SMs of 32 cores each, and of 384 bit wide memory bus delivering 177 GB/sec global memory DRAM bandwidth. The CPU is a quad-core Intel(R) Xeon(R) W3520 clocking at 2.67GHz and having a memory of 8GB. The GCC compiler 4.4.3 is used. The operating system is Linux-Ubuntu kernel 2.6.32-25. The code is compiled with GCC level 3 (-O3) optimization.

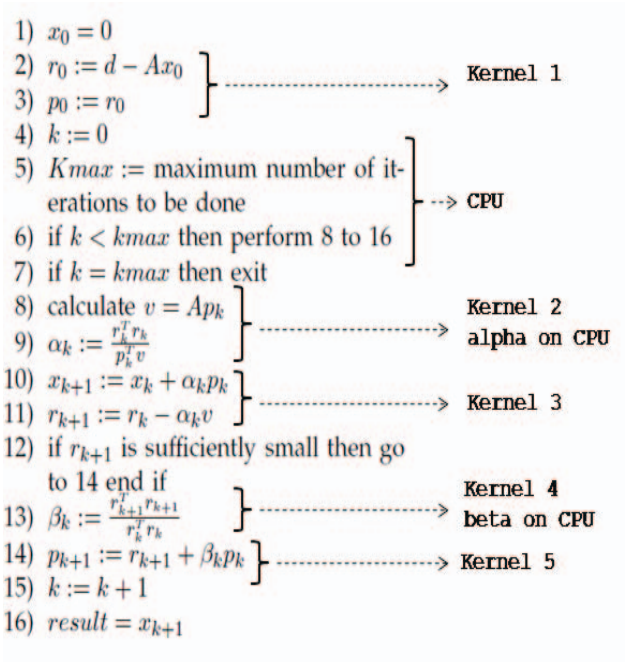

Figure 5. Parallel model of the CG implementation on GPU.

instance the heptadiagonal matrix has additional 2 diagonals compared to the pentadiagonal and consequently the single-threaded CPU-based CG should perform $2N$ operations, whereas in the multiple-threaded GPU-based CG, two extra operations have to be executed by each thread.

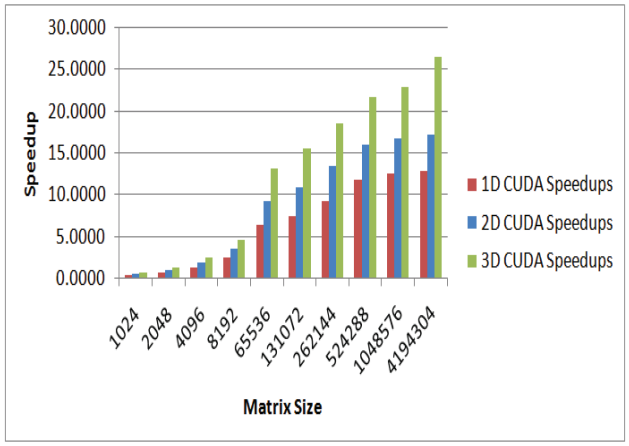

Figure 6. Speedup of the parallel oil reservoir simulators on GPU vis-a-vis their sequential executions on CPU.

B. Experiments

The reservoir used in the experiments is a variation of an example in [2]. The reservoir has homogenous rock properties. The porosity used is 2.5 and anisotropic permeability, $k_x = 150md$, $k_y = 100md$, and $k_z = 100md$. The grid block dimensions are $Ax = 350ft$, $Ay = 250ft$, and $h = 30ft$. The reservoir fluid properties are $B = B^o = 1RB/STB$ and viscosity is 3.5 cp.

To compute the speedup obtained by using GPU, we implemented the simulator in a sequential fashion and run it on CPU only without optimizations and without using any optimized library. The total makespan of the simulator, considering 20 iterations within the CG method, is then measured. We also compute the total makespan of the parallel simulator on the GPU system, considering 20 iterations within the CG method. The total makespan is measured from the time the simulator starts till the time the CG completes 20 iterations. All experiments were done in double precision.

C. Results and Analyses

Figure 6 shows the speedup obtained by the parallel oil reservoir simulators on GPU versus their sequential executions on CPU. For heptadiagonal matrices, we obtained a speedup of 26x versus sequential execution on CPU. Speedups of the oil reservoir simulator for heptadiagonal matrices were the highest compared to pentadiagonal matrices and tridiagonal matrices. This is due to the fact that the CPU version is single threaded and increasing the matrix density increases the number of operations performed. For

VI. CONCLUSION AND FUTURE WORKS

CG method is a mathematical tool used as a solver of systems of linear equations in oil reservoir simulation. In particular, it is used because of its rapid convergence to solution. However, CG method has a performance drawback, mainly due to its sparse matrix-vector multiplication. In this work, we presented a performance evaluation of parallelization of 1-D, 2-D and 3-D, oil-phase reservoir simulators on a GPU system cross 15 Streaming Multiprocessors, making 32 cores each. A simulator generates a structured matrix of either tridiagonal, pentadiagonal or heptadiagonal shape for 1-D, 2-D and 3-D reservoir grids respectively. We described the parallelization strategy of the CG method, used as a linear solver for the simulators. The results show that our parallel implementation scales well with increasing matrix sizes. The CG benefits from fast memory access of the GPU, and an increased locality thanks to DIA format adopted for matrix representation in memory. We obtained a speedup of 13 times faster than the CG CPU-based sequential execution for 1-D oil reservoirs, 17 times faster for the 2-D oil reservoirs, and 26 times faster for the 3-D oil reservoir. This work continues, and the parallel CG linear solver will be integrated into a parallel oil-gas-water (3-phase) oil reservoir simulator under development on the GPU system. This work continues and we are currently exploring the implementation strategies and the performance of 3-phase oil reservoir simulators in both parallel and sequential versions.

ACKNOWLEDGEMENT

This work is supported by a UAE University research grant won by the first author.

REFERENCES

[1] E. Guizzo, "Solving the oil equation," IEEE Spectr., vol. 45, no. 1, pp. 3236, Jan. 2008.

[2] Abou-Kassem, J.H., Farouq Ali, S.M., and Islam, M.R., 2006, "Petroleum Reservoir Simulation: A Basic Approach", Gulf Publishing Company, Houston, TX, USA, 480 pp.

[3] J. Aarnes and K.-A. Lie, "Towards reservoir simulation on geological grid models," in Proc. 9th Eur. Conf. Math. Oil Recovery, Cannes, France, 2004.

[4] C. Farmer, "Flow through porous media and reservoir simulation," in Mathematical Geophysics and Uncertainty in Earth Models. Oxford, U.K.: Univ. Oxford, 2004.

[5] T. Ertekin, J. Abou-Kassem, and G. King, "Basic practical reservoir simulation," in SPE Textbook Series. Richardson, TX: Soc. Petroleum Eng., 2001.

[6] E. Oian, M. S. Espedal, I. Garrido, and G. E. Fladmark, "Parallel simulation of a multiphase/multicomponent flow models," in Lecture Notes in Computational Science and Engineering. New York: Springer-Verlag, 1999, pp. 99-113.

[7] G. Adamson, M. Crick, B. Gane, O. Gurpinar, J. Hardiman, and D. Ponting, "Simulation throughout the life of a reservoir," Oilfield Rev., vol. 8, no. 2, pp. 1627, 1996.

[8] Schlumberger Limited. "Eclipse 2010 Reservoir Engineering Software", http://www.slb.com/services/software/reseng/eclipse2010.aspx, April, 2007

[9] Dogru, Ali h., "From Mega-Cell to Giga-Cell Reservoir Simulation", Saudi Aramco Journal of Technology, Spring 2008.

[10] Dogru, A. H., Li, K. G., Sunaidi, H. A., Habiballah, W. A., Fung, L., Al Zamil, N., and Shin, D., "A Massively Parallel Reservoir Simulator for Large Scale Reservoir Simulation", SPE paper 51886, Presented at the 1999 SPE Reservoir Simulation Symposium, February 1999

[11] Y. Kim, C.C. Swan, and J.-S. Chen, "Performance of parallel conjugate gradient solvers in meshfree analysis of nonlinear continua", Computational Mechanics, 2006.

[12] Piero Lanucara, and Sergio Rovida, "Conjugate Gradients Algorithms: An MPI-OpenMP Implementation on Distributed Shared Memory Systems"

[13] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," nVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-004, Dec. 2008.

[14] Nvida, "CUDA C Programming Guide", PG-02829-001_v5.5, July 2013, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[15] P. Kloos, P. Blaise, and F. Mathey, "Open MP and MPI Programming with a CG Algorithm"

[16] Martin F. Moller, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", Neural Networks, Vol. 6, Issue 4, 25 1993, pp. 525-533.

[17] Lewis and Van de Geijn, "Distributed memory matrix-vector multiplication and conjugate gradient algorithms"', in Proc. Supercomputing'93, Portland, Oregon, pp. 484-492.

[18] Lewis et al, "Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers", in Proc. Scalable High Performance Computing Conference, 1994, pp. 542-550.

[19] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga, "The NAS Parallel Benchmarks", RNR Technical Report RNR-94-007, March 1994.

[20] Jonathon Richard Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", School of Computer Science, Carnegie Mellon University, Edition 1 1/4

[21] Kevin B. Theobald, Gagan Agrawal, Rishi Kumar, Gerd Heber, Guang R. Gao, Paul Stodghill, and Keshav Pingali, "Landing CG on EARTH: A Case Study of Fine-Grained Multithreading on an Evolutionary Path", Proceedings of the ACM/IEEE conference on Supercomputing, pp. 4 - 4, 2000

[22] Fei Chen, Kevin B. Theobald, and Guang R. Gao, "Implementing Parallel Conjugate Gradient on the EARTH Multithreaded Architecture", Sixth IEEE International Conference on Cluster Computing, pp. 459 - 469, 2004

[23] Marty R. Field, "Optimizing a Parallel Conjugate Gradient Solver", SIAM Journal on Scientific Computing, Vol. 19, issue 1, pp. 27 - 37, 1998

[24] Andrzej Jordan and Robert Piotr Bycul, "The Parallel Algorithm of Conjugate Gradient Method", Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers, Vol. 2326, pp. 156 - 165, 2001

[25] Robert Piotr Bycul, Andrzej Jordan, Marcin Cichomski, "A New Version of Conjugate Gradient Method Parallel Implementation", Proceedings of the International Conference on Parallel Computing in Electrical Engineering, pp. 318, 2002

[26] John G. Lewis, Robert A. van de Geijn, "Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms", 1993

[27] John G. Lewis, David G. Payne, "Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers", 1994

[28] Dianne P. O'Leary, "Parallel Implementation of the Block Conjugate Gradient Algorithm", Parallel Computing, Vol. 5, pp. 127 - 139, 1987

[29] Dianne P. O'Leary, "The Block Conjugate Gradient Algorithm and Related Methods", Linear Algebra and its Applications, Vol. 29, pp. 293 - 322, 1980

[30] Haee-Dae Kwon, "Efficient Parallel Implementations of Finite Element Methods Based on the Conjugate Gradient Method", Applied Mathmatics and Computation, Vol. 145, Issue 2-3, 25 December 2003, pp. 869 - 880

[31] Leila Ismail, k. Shuaib, "Empirical Study for Communication Cost of Parallel Conjugate Gradient on a Star-Based Network", ams, pp.498-503, In Proceedings of The 2010 Fourth Asia International Conference on Mathematical/Analytical Modeling and Computer Simulation, 2010

[32] Leila Ismail, "Communication Issues in Parallel Conjugate Gradient Method using a Star-Based Network". 2010 International Conference on Computer Applications and Industrial Electronics (ICCAIE 2010)

[33] Leila Ismail, J. Abou-Kassem, "Toward an Automatic Parallel Load Balanced Distribution Model in Conjugate Gradient Method for One-Dimensional One-Phase Oil Reservoir Simulation". pp.2958-2963, in Proceedings of The IEEE 10th International Conference on Computer and Information Technology, IEEE International Conference on Scalable Computing and Communications (ScalCom 2010)

[34] H. Sudan, H. Klie, R. Li and Y. Saad, "High Performance Manycore Solvers for Reservoir Simulation", 12 th European Conference on the Mathematics of Oil Recovery, 2010

[35] High Performance Computing, Oil and Gas Industry,Manycore and Accelerator-based High-performance Scientific Computing Workshop: http://hpcoilgas.citris-uc.org/node?page=2

[36] Peter N. Glaskowsky, "NVIDIAs Fermi: The First Complete GPU Computing Architecture", a white paper, September 2009

[37] Anirudh Maringanti, Viraj Athavale, and Sachin B. Patkar, "Acceleration of Conjugate Gradient Method for Circuit Simulation Using CUDA", 2009 International Conference in High Performance Computing, pp. 438-444

[38] Mark Harris, "Optimizing Parallel Reduction in CUDA", Technical Report, NVIDIA Developer Technology, 2008

[39] M. Mehri Dehnavi, D. Fernandez, and D. Giannacopoulos, "Enhancing the Performance of Conjugate Gradient Solvers on Graphics Processing Units", IEEE Transactions on Magnetics, pp. 1162-1165, vol.47, issue 5, April 2011

[40] Zhangxin Chen, Guanren Huan, Yuanle Ma, "Computational methods for multiphase flows in porous media", chapter 7, page 265