A044

High Performance Manycore Solvers for Reservoir Simulation

H. Sudan* (ConocoPhillips), H. Klie (ConocoPhillips), R. Li (University of Minnesota) & Y. Saad (University of Minnesota)

SUMMARY

The forthcoming generation of many-core architectures compels a paradigm shift in algorithmic design to effectively unlock its full potential for maximum performance. In this paper, we discuss a novel approach for solving large sparse linear systems arising in realistic black oil and compositional flow simulations. A flexible variant of GMRES (FGMRES) is implemented using the CUDA programming model on the GPU platform using the Single Instruction Multiple Threads (SIMT) paradigm by taking advantage of thousands of threads simultaneously executing instructions. The implementation on the GPU is optimized to reduce memory overhead per floating point operations, given the sparsity of the linear system. FGMRES relies on a suite of different preconditioners such as BILU, BILUT and multicoloring SSOR. Additionally, the solver strategy relies on reordering/partitioning strategies algorithms to exploit further performance. Computational experiments on a wide range of realistic reservoir cases show a competitive edge when compared to conventional CPU implementations. The encouraging results demonstrate the potential that many-core solvers have to offer in improving the performance of near future reservoir simulations.

Abstract

With the advent of the processor technology revolution, many-core computing with Graphics Processing Units (GPUs) provides enormous processing capability suitable for large scale engineering and scientific applications. The forthcoming generation of many-core architectures compels a paradigm shift in algorithmic design to effectively unlock its full potential for maximum performance. In this paper, we discuss a novel approach for solving large sparse linear systems arising in realistic black oil and compositional flow simulations. A flexible variant of GMRES (FGMRES) is implemented using the CUDA programming model on the GPU platform using the Single Instruction Multiple Threads (SIMT) paradigm by taking advantage of thousands of threads simultaneously executing instructions. The implementation on the GPU is optimized to reduce memory overhead per floating point operations, given the sparsity of the linear system. FGMRES relies on a suite of different preconditioners such as BILU, BILUT and multicoloring SSOR. Additionally, the solver strategy relies on reordering/partitioning strategies algorithms to exploit further performance. Computational experiments on a wide range of realistic reservoir cases show a competitive edge when compared to conventional CPU implementations. The encouraging results demonstrate the potential that many-core solvers have to offer in improving the performance of near future reservoir simulations.

1. Introduction

Due to transistor physical limitations, the performance of single core is levelling off and progressively being replaced by many-core solutions. This implies a radical paradigm shift in the design of software for engineering and scientific applications. There is no doubt that this paradigm shift will have important computational implications in the oil industry and in particular, reservoir simulation.

GPU technology has already provided many success stories in the Oil and Gas Industry, more specifically in geophysical applications (B. Deschizeaux et al., 2008; Micikevicius, 2009). This is primarily due to the high level of parallelism implied by the processing of large sets of independent seismic data using the SIMD (Single Instruction Multiple Data) paradigm.

In the case of reservoir simulation, the success of efficient GPU implementations is limited to spatial and temporal dependencies established by the discretized equations governing flow phenomena. Traditionally, each simulation component may require a different approach to parallelization based on the physics of the problem (e.g., black-oil, compositional, thermal), the numerical formulation (e.g. degree of implicitness, type of spatial discretization and meshing), the input data (e.g., reservoir geometry, heterogeneity) and user supplied options (e.g., timestep control, flash calculations). Challenges associated with the parallelization of reservoir simulation applications have been extensively discussed in several papers, among recent ones (DeBaun et al., 2005, Al-Shaalan et al., 2009; Dogru et al., 2009). Authors of the present paper (Wang et al., 2009) have already performed some preliminary investigations to analyze GPU scalability of isolated linear systems arising in black-oil and compositional simulations. The present work represents updated numerical results and algorithm extensions to a currently submitted paper (Li et al., 2010).

In this paper, we propose a many-core GPU implementation for solving large sparse linear systems arising in realistic black-oil and compositional flow simulations. The present paper is structured as follows. In the following section, we provide a brief description of the GPU architecture to facilitate the understanding of the solver implementation. We then provide a discussion of the solver implementation that relies on an efficient implementation of matrix-vector products and construction and application of the preconditioner. The discussion is followed by benchmarks illustrating the performance of the parallel solver and its impact on accelerating simulations on the well known SPE10 and three field cases. These benchmarks are established using state-of-the-art CPU and GPU technology. We conclude the present work with some highlighting remarks and future lines of effort. Henceforth, we will be referring to multicore architecture of CPU as just “CPU” and the many-core GPU as “GPU” unless stated otherwise.

2. The GPU/CUDA Many-core Paradigm

The GPU architecture is based on a highly multithreaded SIMD paradigm. More specifically, the same program called kernel, is running on GPU cores that is executed by each GPU thread with different data. The CPU launches the sequence of kernels to allow the computationally intensive operations to be executed in the GPU.

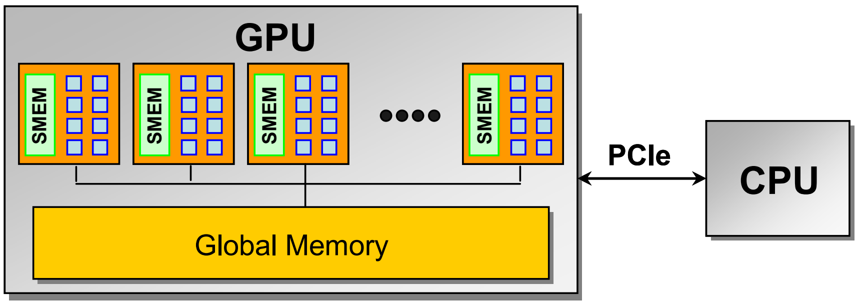

Figure 1. GPU hardware overview.

Figure 1 illustrates the basic GPU architecture and its connection with the CPU. The CUDA programming model provides the level of abstraction required to be able to handle hierarchy of threads, memory and synchronization instructions.

The Tesla 20-series GPU, also commercially codenamed *Fermi* (Tesla C2050) that was used in our numerical experiments, contains 14 streaming multiprocessors, each with 32 scalar processors (for a total of 448 computing cores), 32K 32-bit registers, and 48 KB of shared memory and global memory up to 6GB. Global memory accesses for the 20-seriers family of Tesla GPUs are cached. There is an L1 cache for each multiprocessor and an L2 cache shared by all multiprocessors. This allows accelerating algorithms such as the sparse matrix-vector-products that are characterized by indirect and irregular data accesses. Each multiprocessor is capable of handling 1536 threads, so the total number of threads that can handle the GPU is 14x1536=21,504 threads. To manage this large population of threads efficiently, the threads are organized in thread blocks in groups of 32 called warps. Consequently, threads within a thread block can communicate via shared memory, or use shared memory as a user-managed cache since shared memory latency is two orders of magnitude lower than that of the global memory. A barrier primitive is provided so that all threads in a thread block can synchronize their execution.

The stream processors are fully capable of executing integer and double precision floating point arithmetic. These processors have access to global device memory. Memory latency is hidden by executing thousands of threads concurrently. Register and shared memory resources are partitioned among the currently executing threads. There are two major differences between CPU and GPU threads. First, the GPU threads are lightweight and non-persistent compared to CPU threads. Second, while CPUs execute efficiently when the number of threads per core is small (often one or two), GPUs achieve high performance when thousands of threads execute concurrently.

The connection between CPU memory and GPU memory is established through a fast PCI express-16X point-to-point link. In general this memory transfer is marginal when the GPU computing time governs most of the overall computing time (Wang et al. 2009). Further technical information on both GPU architecture and CUDA are available in (Lindholm et al., 2008; Nickolls et al., 2008).

3. General Description of the Many-core GPU Solver

Our GPU solver implementation relies on a flexible variant of GMRES, namely FGMRES (Saad 2003). The FGMRES solver allows for using variable preconditioning in each of the iterations and can handle nested type of solutions in a more stable fashion. The implementation relies on CUBLAS library (Deschizeaux et al., 2008) kernel calls, a specific adaptation of a set of array operations to carry out Krylov basis orthogonalization, sparse matrix-vector products (*Spmv*) operations and the preconditioner construction and application. CUBLAS is a highly optimized BLAS library for GPUs specially written in CUDA.

In terms of the implementation, the Spmv and preconditioner are the tasks responsible for dictating the overall complexity in each of the solver iterations and its computational efficiency is critical for achieving high overall solver performance on the GPU. Additionally, sparsity of the matrix generally induces irregular memory access patterns. Hence mapping the number of threads to optimally reduce memory overhead is essential in our CUDA implementation.

3.1 GPU Sparse Matrix-Vector Product Kernel

We use the compress storage row (CSR) to represent the matrix system (Saad, 2003). That is, a floating point array is used to store all the non-zeros; one integer array contains the column index of each non-zero and another integer array contains pointers of the starting position of each row.

A straightforward parallel implementation can be achieved by partitioning the matrix by rows and distribute them among different thread blocks. However, this type of partitioning may induce poorly coalesced transactions since each thread will be fetching data in shared memory. On the other hand, if locations being accessed are sufficiently close to each other, more thread operations can be coalesced by hardware to achieve greater memory efficiency.

A better implementation is to assign half-warp (16 consecutive threads) per row. That is, the first half warp works on row one, second half warp works on row two and so forth. Since all threads within a half warp access nonzero elements of a given row the chance of coalescing should be much higher. Higher performance can be actually achieved if, in addition, these nonzero elements are accessed consecutively in memory. Since each row is multiplied by a vector to get an entry in the output vector, the operation reduces to perform a parallel dot product. Since the input vector is read-only, the best strategy is to place this vector in the texture memory and access it by texture fetching. Upon completion of the set of parallel dot products, partial results in shared memory can be summed via a fan-in scheme (Bell and Garland, 2009; Baskaran and Bordawekar, 2009; Buatois et al., 2009).

Based on our experiments, the performance of the second implementation is much higher than the first one for matrices in which the number of non-zeros per row is large. However, for matrices which have a small number of non-zeros per row, the performance of the first implementation is close to the second one. For some cases, it can even outperform the second implementation.

A more robust implementation of Spmv kernels is to use the JAD (JAgged Diagonal) format. The Spmv kernel in this format is observed to provide high performance for matrices with high or low number of nonzeros per row. The only overhead incurred in using this format is to permute matrix rows according to the number of non-zeros of each row. The JAD format and the implementation of matrix-vector product in this format are discussed in detail in (Saad, 2003).

3.2 GPU Preconditioner Implementation

We consider a suite of different block preconditioners on the GPU. Our primary intention is to evaluate their suitability to exploit maximum performance on the GPU and eventually decide smart strategies to select them according to the simulation problem. This last aspect will be discussed in near future works.

The use of block strategies such as block Jacobi allows for assigning each set of block coefficients to a thread block. We consider the following preconditioning approaches: (a) BILU(k). i.e., BILU with a level of fill-in according to k, (b) BILUT, i.e., BILU with threshold that basically stands for a BILU with both a fill-in and drop tolerance strategy and, (c) multicoloring SSOR (MC-SSOR), that is based on a greedy reordering strategy to compute the minimum number of colours possible from the matrix connectivity graph. The latter approach is nothing else than a generalization of the red-black colouring used to generate inexpensive Schur complement formulations associated to the problem. As in the case of the Smpv algorithm, the number of threads per coefficient block is approximately given by the size of the problem divided by the number of colours times the number of thread blocks. Additionally, a sparsification strategy is applied to discard small entries during the graph colouring process. In the application phase of the preconditioner, a few block SSOR sweeps is performed to reduce residuals at each FGMRES iteration. Further technical details about the implementation of these preconditioners are available elsewhere (Li and Saad, 2010; Li et al., 2010).

Given the SIMD paradigm in the GPU, both the setup and application phases of each of the aforementioned preconditioners lead to local solutions without any (BILU variants) or marginal (in the case of multicolour SSOR) data dependency. The global solution is obtained by adding contributions from each local linear system. Despite the high parallelism, the quality of the preconditioner deteriorates as the number of blocks increases. This deterioration reflects to an increasing number of iterations. For a wide range of problems it is challenging to determine the optimal number of GPU threads required to achieve maximum parallelism while maintaining the effectiveness of the preconditioner. We found that by running a few short inspectional runs with different number of threads were useful to estimate an optimal number to deliver the best possible performance. For reservoir cases consisting of around 500,000 gridblocks, a value of 512 thread blocks was close to the optimal number.

In order to achieve maximum parallel efficiency in the suite of BILU preconditioners, it is fundamental to reduce irregular memory accesses and balance the load of computational work among processors. In the context of graphs, the goal is to simultaneously minimize the distance between nodes (matrix entries) and the number of edge cuts upon assigning nodes to different processors. Seeking optimal partitioning and reordering of variables to achieve maximum parallel performance has been actually subject of intensive research (Saad, 2003; Shuttleworth et al., 2009).

Among several reliable graph partitioning software, we use METIS to generate a balance partitioning of tasks among processors (Karypis and Kumar, 1998). To minimize matrix bandwidth and, therefore, the degree of fill-in in BILU type of implementations, we rely on the Reverse Cuthill-McKee (RCM) algorithm (George and Liu, 1981). For large problems, the cost of using both algorithms may be considerable high relative to the solution process. However, since linear systems are generally solved hundreds of thousand of times during the span of the simulation, this cost can be effectively amortized. Generally, both METIS and RCM are required to be used a small fractional number of times with respect to the total number of timesteps in the whole simulation. We have implemented simple indicators to automatically perform a fresh METIS or RCM reordering after long intervals of simulation. In order to be able to perform partitioning or reordering of a given linear system at any timestep, we only require storing an index permutation vector.

METIS partitioning generates denser diagonal blocks of similar size with respect to the original matrix. If additionally RCM reordering is applied before METIS, then a more compact banded local system will be obtained in each block. Note that the use of both partitioning and reordering operations increases the chances to move more entries inside each diagonal block but, on the other hand, this may negatively affect the original degree of diagonal dominance and thereby negatively impact the solver scalability. We have developed a set of automatic criteria to select the use of RCM when impacts positively in the linear solver convergence.

4. Numerical Experiments

We consider 4 different simulation cases in our performance study. These cases include an oil-water case given by the SPE10 synthetic case (Christie and Blunt, 2001) and 3 field cases: a black-oil case (Case A) and 2 compositional cases (Cases B and C) involving 5 and 8 components, respectively. All cases are run using the IMPES option with implicit well treatment, so the benchmarks are based on the solution of the pressure system. Succinct description of these cases is summarized in Table 1. The four cases yield linear systems that vary significantly in size, sparsity and conditioning. As the number of mass balance equations increases, we observe that the fraction of the solver responsible for the total CPU time decreases. In the case of compositional simulation (cases B and C) a fraction of about 20% is spent in the thermodynamics of the phase behaviour calculation.

Table 1. Benchmark cases.

Case	Size	Model type	% Solver	N. wells	N. Timesteps	Most relevant characteristics
SPE10	60x220x85	Oil-water	90%	5	1423 (50 days)	Highly heterogeneous
A	140x230x44	Black-Oil	81%	83	2524 (34 years)	Waterflooding, highly heterogeneous
B	128x155x19	5 comp.	75%	445	19429 (35 years)	Highly faulted, water injection
C	257x228x21	8 comp.	72%	124	1641 (5 years)	Highly compartmentalized

We first focus our attention on comparing the performance of the FGMRES iterative solver using BILU(0), BILUT and MC-SSOR on a single core CPU and on the GPU. The fill-in and drop threshold for BILUT were set 30 and 1.D-4, respectively. The number of SSOR iterations was set to 2. These preconditioners can be considered the main work-horse options in any reservoir simulator solver available today. In all cases to be shown, the FGMRES solver was assumed to converge when a relative residual less or equal to 10^{-4} was achieved.

We hardware specifications are the following:

- Dual Intel Nehalem (Intel Xeon X5570, 2.93 Ghz, 8MB L3 Cache), 16GB RAM
- NVIDIA Tesla C2050 (installed with Intel Intel Nehalem X5570), 448 cores, 3.22 GB RAM

The Intel Nehalem family architecture is considered the fastest multicore CPU available nowadays. All runs are carried out on Linux. Our in-house simulator was compiled using the –O2 option and linked with the GPU Solver already compiled as a form of a library. The resulting object code was run it on the two hardware platforms mentioned above. Since we are mainly concerned with maintaining accuracy in real field simulations, we restrict our analysis to a double precision version of the GPU solver library. Hence, the compiler option for the C interface to CUDA was set to –arch sm_20 in order to enable the double precision option and global memory cache.

Tables 2, 3 and 4 shows the solver performance using the BILU(0), BILUT and MC-SSOR preconditioners on the GPU for isolated systems obtained from the 4 cases considered. The BILU set of preconditioners rely on METIS partitioning. We are omitting the RCM reordering since it did not show a major influence in the CPU timings for the cases considered. We can note that the BILUT becomes more effective than BILU(0) as the problems sizes increases. The MC-SSOR strategy outperforms these two BILU strategies in all cases. Most the gain here is due to the inexpensive construction of the preconditioner for the MC-SSOR. In each case, we can observe that the CPU Spmv operation is practically negligible compared to the application of the preconditioner.

The left side of Figure 2 compares the performance of the GPU-based solver using the BILU(0) preconditioner relative to the CPU-based solver using the ILU preconditioner upon completion of the simulation. The right side of Figure 2 shows the relative simulation performance using these two solvers. Note that the solver time represents a fraction of the whole simulation (see column 4 on Table 1). The GPU BILU(0) solver shows a gain factor of approximately *3x* with respect to Intel Nehalem. It is worth to mention that the GPU BILU solver is weaker than the ILU sequential implementation as the iterative solver generally takes more iterations. Nevertheless, GPU solver is still able to outperform, since thousands of GPU threads are exploited in each of the FGMRES iterations.

Table 2. Comparative performance assessment using the GPU BILU(0) solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	METIS (s)	N. Iterations
SPE10	BILU(0)	1.06	3.54	0.35	0.58	5.53	1.32	168
A	BILU(0)	0.25	0.58	0.06	0.30	1.19	1.03	58
B	BILU(0)	0.08	0.23	0.03	0.09	0.43	0.20	55
C	BILU(0)	0.08	0.08	0.01	0.02	0.19	0.14	22

Table 3. Comparative performance assessment using the GPU BILUT solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	METIS (s)	N. Iterations
SPE10	BILUT	1.06	2.88	0.27	0.56	4.77	1.32	120
A	BILUT	0.82	0.75	0.06	0.33	1.96	1.03	58
B	BILUT	0.28	0.50	0.03	0.10	0.91	0.20	51
C	BILUT	0.27	0.11	0.01	0.03	0.42	0.14	19

Table 4. Comparative performance assessment using the GPU CS-SSOR(2) solver.

Cases	Prec	Prec. Setup (s)	Prec. Appl. (s)	Matvec (s)	Remainder (s)	Total (s)	N. Iterations
SPE10	MC-SSOR	0.02	1.88	0.16	0.77	2.83	91
A	MC-SSOR	0.02	0.81	0.06	0.26	1.15	59
B	MC-SSOR	0.01	0.22	0.03	0.09	0.35	50
C	MC-SSOR	0.01	0.13	0.02	0.05	0.21	34

The results shown at the right side of Figure 2 are a immediate consequence of Amdahl's Law when a fraction of the original code is only parallelized, namely, the performance improvement PI is given by PI=1/(s + p), where s represents the sequential fraction and p represents the parallel fraction of the simulation code. For instance, the solver component of Case C represents 72% of the total time and the GPU solver is about 2.8 times faster than the CPU Intel Nehalem, thus the expected performance improvement in the simulation is approximately given by PI = 1/(.28 + .72/2.8) ~ 1.8x. This can be seen clearly reflected in the right side of Figure 2.

5. Conclusions and Future Work

We have evaluated the current capabilities that many-core GPU on a set of realistic black-oil and compositional reservoir scenarios. Despite that the study focused only on the solver component of the simulator and that there is still room for further improvements in the proposed implementation, the results provide clear indication of the enormous potential that many-core GPU can offer in near future simulation studies.

We observed that with a moderate effort in the solver kernel, the GPU can outperform by almost ~3x the serial implementation on the Intel Nehalem. This improvement resulted in a noticeable improvement in the simulation of field cases, in the order of 1.8x-2.4x. This gain seems to be more significant as the problem size increases and more regular computations are fitted into the SIMD paradigm entailed by GPUs. Potential for achieving higher simulation performance seems to be more promising using MC-SSOR rather than block ILU type of approaches. Further attention will be devoted to analyze the interplay of the MC-SSOR preconditioner upon sparsification and other reordering strategies. Additional performance can be particularly exploited in cell-wise type of computations such as those involved in physical properties, phase behavior and other vector computations related to the solution process.

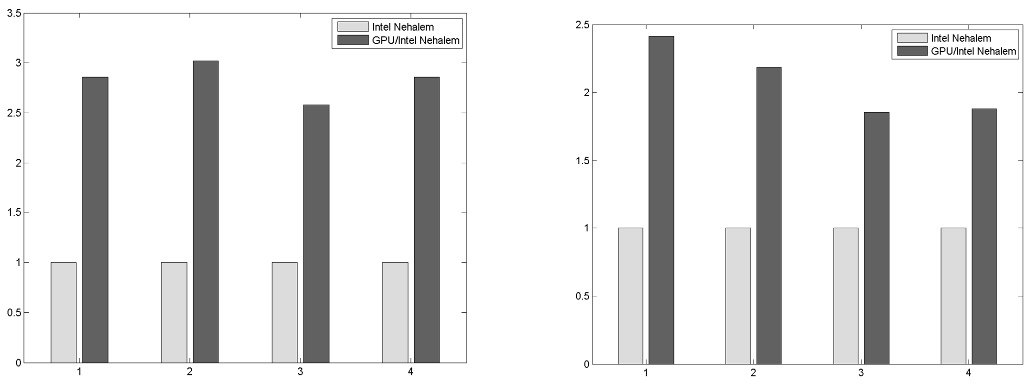

Figure 2. Relative performance for the solver (left) and full simulation (right) on the CPU and GPU.

Despite that the results on GPU look comparatively impressive with respect to the serial implementation on a state-of-the-art multicore CPU platform, the current factor of 3x may not be enough to outperform highly scalable parallel algorithms on multicore CPU. However, from our perspective, the advance in many-core hardware technology has the potential to surpass the current capabilities of the multicore CPU technology for SIMT type of tasks. We have observed tremendous benefit exploiting dense and particular sparse BLAS operations (such as our current Spmv implementation) in our performance studies.

Having the GPU as a specialized accelerator device attached to the multicore CPU, there are clear opportunities to develop hybrid algorithms that can exploit different levels of parallel granularity in both of the architectures to effectively handle multiple simultaneous tasks (e.g., multistage preconditioning, upscaling, well allocation factor computations, reduced order model generation). We believe that as the parallelism implied by many-core technology becomes more widely accepted, large and complex scale simulations will be part of day-to-day desktop computations.

Acknowledgements

The authors would like to thank ConocoPhillips Company for allowing the publication of the current work. This work is also supported by DOE under grant DE-FG 08ER25841. The authors want to thank NVIDIA for facilitating access to the Fermi architecture.

References

T. M. Al-Shaalan, A. H. Dogru, H. Klie and M.F. Wheeler. *Studies of Robust Two-Stage Preconditioners for the Solution of Fully Implicit Multiphase Problems*. SPE Reservoir Simulation Symposium, SPE paper N. 118722, The Woodlands, Texas, Feb. 2-4, 2009.

S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, E.S. Quintana-Orti and G. Quintana-Orti. *Exploiting the Capabilities of GPUs for Dense Matrix Computations.* Technical Report ICC 01-11-2008, Departamento de Ingenieria y Ciencias de Computadores, Castellon, Spain, 2008.

M. M. Baskaran and R. Bordawekar. *Optimizing Sparse Matrix-Vector Multiplication on GPUs*. Technical report, IBM Research, 2009.

N. Bell, M. Garland. *Implementing sparse matrix-vector multiplication on throughput-oriented processors.* SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 1-11, 2009.

L. Buatois, G. Caumon, B. Levy. *Concurrent number cruncher: A GPU Implementation of a General Sparse Linear Solver.* International Journal of Parallel, Emergent and Distributed Systems, 24: 205-223, 2009.

M.A. Christie and M.J. Blunt. Tenth *SPE Comparative Solution Project: A Comparison of Upscaling Techniques*. SPE Reservoir Simulation Symposium, SPE paper N. 72469, Houston, Feb. 11-14, 2001.

B. Deschizeaux and J-Y. Blanc. *Imaging Earth's Subsurface using CUDA.* GPU Gems 3, Chapter 38, H. Nguyen editor, NVIDIA Corporation, 831-850, 2008.

A.H. Dogru, L.S.K. Fung, U. Middya, T.M. Al-Shaalan, J.A. Pita, K. Hemanth, Kumar, H.J. Su, J.C.T. Tan, H. Hoy, W.T. Dreiman, W.A. Hahn, R.Al-Harbi, A. Al-Youbi, N.M. Al-Zamel. M.Mezghani, T. Al-Mani. *A Next Generation Parallel Reservoir Simulator for Giant Reservoirs*. SPE Reservoir Simulation Symposium, SPE paper N. 119272, The Woodlands, TX, Feb, 2009

J.A. George and J.W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewwod Cliffs, N.J., 1981.

G. Karypis and V. Kumar. *A Parallel Algorithm ofor Multilevel Graph Partitioning and Sparse Matrix Ordering*. Journal of Parallel and Distributed Computing, 48: 71-95, 1998.

R.Li, Y. Saad. *Preconditioned Iterative Linear Solvers on GPUs*. In preparation, 2010.

R. Li, H. Klie, H. Sudan, and Y. Saad. *Towards realistic reservoir simulations on manycore platforms*. SPE Journal, 2010. Submitted.

E. Lindholm, J. Nickolls, S. Oberman and J. Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. Micro, IEEE. 28:39-55, 2008.

P. Micikevicius. *3D finite difference computation on GPUs using CUDA*. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 79-84, 2009.

J. Nickolls, I.Buck, M. Garland and K. Skadron. *Scalable Parallel Programming with CUDA*. Queue, ACM. 6:40-53, 2008.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, Society for Industrial and Applied Mathematics, Philadelphia, 2003.

M. Wang, H. Klie, M. Parashar and H. Sudan. *Solving Sparse Linear Systems on NVIDIA Tesla GPUs*. Lecture Notes in Computer Science, Springer, 5544: 864-873, 2009.