ARTICLE IN PRESS

Computer Physics Communications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A CUDA based parallel multi-phase oil reservoir simulator

Ayham Zaza [a,*], Abeeb A. Awotunde [b], Faisal A. Fairag [c], Mayez A. Al-Mouhamed [a]

[a] Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia
[b] Petroleum Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia
[c] Mathematics and Statistics Department, King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia

ARTICLE INFO

Article history:
Received 2 May 2015
Received in revised form
2 January 2016
Accepted 20 April 2016
Available online xxxx

Keywords:
Parallel computing
Oil reservoir simulation
CUDA
Krylov iterative solvers

ABSTRACT

Forward Reservoir Simulation (FRS) is a challenging process that models fluid flow and mass transfer in porous media to draw conclusions about the behavior of certain flow variables and well responses. Besides the operational cost associated with matrix assembly, FRS repeatedly solves huge and computationally expensive sparse, ill-conditioned and unsymmetrical linear system. Moreover, as the computation for practical reservoir dimensions lasts for long times, speeding up the process by taking advantage of parallel platforms is indispensable. By considering the state of art advances in massively parallel computing and the accompanying parallel architecture, this work aims primarily at developing a CUDA-based parallel simulator for oil reservoir. In addition to the initial reported 33 times speed gain compared to the serial version, running experiments showed that BiCGSTAB is a stable and fast solver which could be incorporated in such simulations instead of the more expensive, storage demanding and usually utilized GMRES.

1. Introduction

From food production, power generation to transportation systems and almost every other aspect of daily life, our modern society continues to ask for more and more energy with oil being the number one resource that addresses such heavily increasing demands. Despite the huge technological advances in oil industry, recovering remaining oil is limited by our knowledge and understanding of the oil reservoir [1]. Reservoir simulation is a challenging mathematical process that models fluid flow and mass transfer in porous media and aims at providing vital information about reservoir structure, production rate, cost management, optimal well placement and many others. The simulation process requires large amount of memory storage and computations. Moreover, as the computation for practical reservoir dimensions may last for long times, speeding up the process by taking advantage of parallel computing is indispensable.

Just like many other complex systems in nature, the reservoir behavior could be modeled and described using set of partial differential equations (PDEs) that mimics how the entire system evolves in time, space or both. For many practical scenarios, obtaining a closed form analytical solution for the governing (PDEs) is infeasible. For that reason various discretization schemes are employed to approximate the solution while maintaining an accepted level of stability and accuracy. Such approximations lead to a large sparse system of algebraic equations that needs further to be solved. Regardless of the discretization scheme, mesh type or the solution approach, the Forward Reservoir Model (FRS) will eventually infer reservoir structure and configurations through the estimation of various essential spatial properties. Such estimation is later utilized to either quantify the production rate in the development of new fields or to instantiate another process, namely, the inverse model or history matching.

This paper is organized as follows: Section 2 starts by describing the forward reservoir simulation process and its associated computational model. Before presenting various considerations for the parallel implementation in Section 2.3, and because it accounts for more than 67% of the computational complexity, algebraic linear solvers are reviewed in Section 2.2 and criteria for selecting a solver that suites FRS was established and later tested. After that, Section 2.4 summarizes various aspects of many-core machines and CUDA programming model. Section 3 describes in detail the methodology while Section 4 discusses the findings and provides conclusions on the obtained results.

* Corresponding author.
E-mail addresses: ayham@kfupm.edu.sa (A. Zaza), awotunde@kfupm.edu.sa (A.A. Awotunde), ffairag@kfupm.edu.sa (F.A. Fairag), mayez@kfupm.edu.sa (M.A. Al-Mouhamed).

http://dx.doi.org/10.1016/j.cpc.2016.04.010
0010-4655/

2. Background

2.1. Forward Reservoir Model

The implemented (FRS) models a 3D flow process of two immiscible phases (water, oil) and accounts for various physical properties in the flowing medium like permeability, porosity, oil pressure, water saturation as well as the interacting forces such as gravity and capillary. Permeability is the capacity of the rock to transmit fluid through its connected pores when the same fluid fills all the interconnected pores [2]. A porous medium is a solid containing void spaces (pores), connected or unconnected, dispersed within it in either a regular or random manner. And porosity is the ratio of the volume of the pores to the total bulk volume of the media [3]. As will be detailed later, our developed simulator will have isotropic permeability distribution, supports heterogeneous geometry,[1] handles different boundary conditions and takes into account various well constrains.

In the analysis stage, the mass balance equation for every phase is constructed and the associated velocities are expressed using Darcy's law. Mathematically, the mass balance equation could be derived as:

$$-\nabla\cdot\left(\rho\vec{u}_f\right)+\rho_f\frac{q_f^{well}}{V_b}=\frac{\partial}{\partial t}\left(\phi\rho_f S_f\right), \tag{2.1}$$

and Darcy Law is given by:

$$\vec{u}_f=-\frac{1}{\mu_f}k_{rf}\underline{\underline{k}}\cdot\left(\nabla p_f+\gamma_f Z\right), \tag{2.2}$$

where, the subscript $f\epsilon$[oil (o) , water (w)], $\vec{u}_f$ is velocity vector, ρ_f the density, q_f^{well} the flow rate, V_b is the bulk volume, ϕ the porosity of the medium, and S_f is phase saturation. $\underline{\underline{k}}$ represents the absolute permeability tensor of the medium, k_{rf} is the relative permeability of phase f, μ_f is the viscosity of phase f, $\vec{u}_f$ is the velocity of phase f, p the applied pressure drop, Z is the depth of the reservoir and γ is the specific gravity of the fluid.

Expanding Eq. (2.1) using suitable flow units, and after substituting the velocity from Eq. (2.2) we obtain the following equations for each phase,

$$\frac{\partial}{\partial x}\left(\frac{\beta_c k_{ro}K_xA_x}{\mu_o B_o}\frac{\partial p_o}{\partial x}\right)\Delta x+\frac{\partial}{\partial y}\left(\frac{\beta_c k_{ro}K_yA_y}{\mu_o B_o}\frac{\partial p_o}{\partial y}\right)\Delta y+\frac{\partial}{\partial z}\left(\frac{\beta_c k_{ro}K_zA_z}{\mu_o B_o}\frac{\partial \varphi_o}{\partial z}\right)\Delta z+q_{osc}=\frac{V_b}{\alpha_c}\frac{\partial}{\partial t}\left(\frac{\emptyset S_o}{B_o}\right). \tag{2.3}$$

$$\frac{\partial}{\partial x}\left(\frac{\beta_c k_{rw}K_xA_x}{\mu_w B_w}\frac{\partial p_w}{\partial x}\right)\Delta x+\frac{\partial}{\partial y}\left(\frac{\beta_c k_{rw}K_yA_y}{\mu_w B_w}\frac{\partial p_w}{\partial y}\right)\Delta y+\frac{\partial}{\partial z}\left(\frac{\beta_c k_{rw}K_zA_z}{\mu_w B_w}\frac{\partial \varphi_w}{\partial z}\right)\Delta z+q_{wsc}=\frac{V_b}{\alpha_c}\frac{\partial}{\partial t}\left(\frac{\emptyset S_w}{B_w}\right). \tag{2.4}$$

Two more equations are then needed to close the system. In the two-phase system considered in this work, we require that:

$$p_c=p_o-p_w, \tag{2.5}$$

$$S_o+S_w=1, \tag{2.6}$$

where, B is the formation volume factor, α_c and β_c are constants, k_{ro} and k_{rw} are the relative permeability for oil and water respectively. Finally p_c is the capillary pressure.

[1] Those are properties of the porous media; Isotropic means the permeability is constant in all directions, i.e. it does not exhibit directional bias. Heterogeneous: porosity is changing with location.

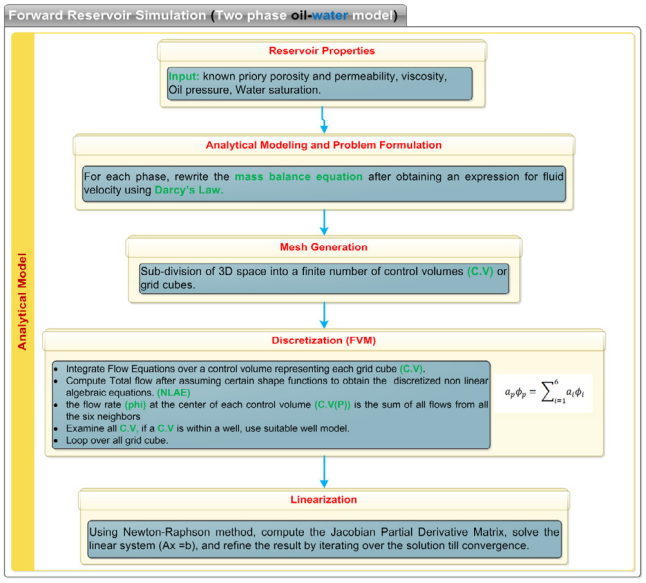

Fig. 1. General description for the forward reservoir simulation model.

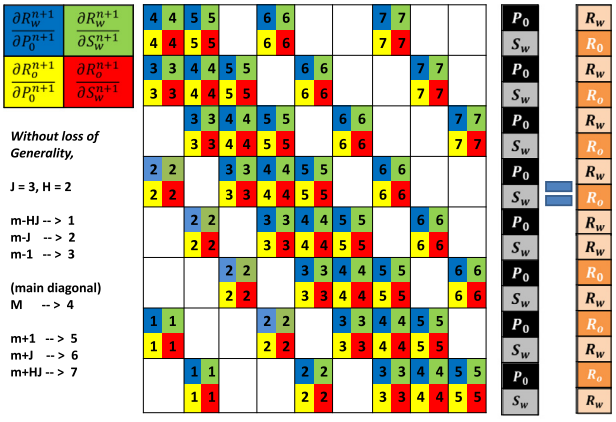

Fig. 2. Sample snapshot of the assembled linear system for FSR, (J, I and H): is the maximum number of steps in the z, x and y directions, respectively.

Natural grid indexing was utilized and the above equations were discretized using the finite volume method [4] on a structured grid. After the discretization step, the resulting system of nonlinear algebraic equations for each phase is then written in terms of its corresponding residual equation R_o and R_w. System linearization is achieved by applying Newton Raphson Method. The method repeatedly refines a nearby approximation after solving a system of linear equations with their coefficient matrix represented by the Jacobian of the Residuals. The Jacobian is obtained for each phase by deriving the residual equation with respect to both P_o and S_w at each grid point and all its neighbors. Fig. 1 shows a general description for the FRS process. Without loss of generality, Fig. 2 shows how the final assembled system for a sample 3D grid looks like. For more details on the mathematical formulation please see [5].

Fig. 3 presents the computational aspect of the simulator. The forward model consists of three main iterations, namely L1, L2 and L3 as well as a fourth optional one L4. They are detailed as follows:

- The outer most (Loop L1): is the temporal loop which repeats the whole simulation for different time steps and usually measured in days.
- The middle iteration (Loop L2): is Newton iteration that achieves the linearization. During this iteration the resulted sparse linear system of the form $Ax = b$ is solved. Solving a

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

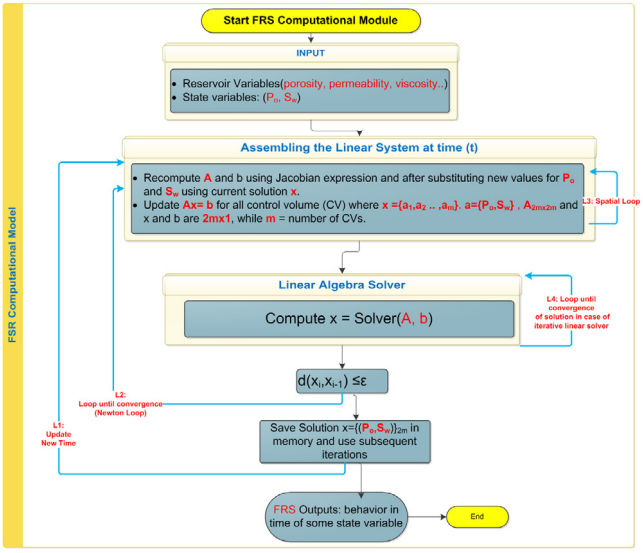

Fig. 3. General computational scheme for the Forward Oil-Black model: When assembling the linear system. All grid points are visited. Newton iteration repeatedly solves the system of linear equations formed in the grid iteration.

linear system accounts for about 67% of the total simulation time.

- The most inner one (Loop L3): is the spatial loop that visits all system grid cubes and form the corresponding nonlinear system to be later linearized, solved and refined during the middle iteration (L2).
- Optional (Loop L4): this loop is available if iterative methods are used to solve the linear system.

For a three-dimensional problem and two simulated phases discretized using finite volume method, it is clear that the maximum number of nonzero elements at each row is 14. Let N be the total number of grid cubes, then for a two-phase system, the size of the Jacobian matrix is $(2N \times 2N)$, and the total number of nonzero elements is at most $(14 \times 2N)$. As a result, the fraction of nonzero elements in the system is at most

$$\frac{14 \times 2N}{2N \times 2N} = \frac{7}{N}.$$

It is obvious that special care should be taken when selecting a suitable solver to accompany FRS implementation, especially for practical dimensions ($N = 10^6$), as most of the operations on the zero elements are not necessary and should be avoided to reduce the computational complexity. Not only is the resulting Jacobian matrix is sparse, but also it is unsymmetrical, ill-conditioned and has a special Hepta-diagonal structure. This also influences the selection or implementation of any linear solver. To reduce storage requirements, sparse matrix representations have evolved to efficiently identify, operate on, and manipulate all nonzero matrix elements. As opposed to dense matrices, a sparse matrix is a matrix in which most of the elements are zero. The reader is referred to [6–11] for more in depth review of sparse matrices on CPU, Multicore and Many-Core devices, their representations and comparisons, [12,13] for studies dedicated to diagonal matrices, and [11,14–16] for blocking restructuring techniques.

2.2. Linear solvers review

Although a clear boundary between the two famous classifications of Linear Solvers is very blur as indicated by [17] and since they are context specific, one can still utilize the classic (direct, iterative) taxonomy and its subcategories to provide better rationalization when picking up the right solver for any application of interest.

If the coefficient matrix (A) is non-degenerate, non-singular, direct solvers in the absence of rounding errors, offer the exact solution in finite steps with robust and predictable behavior without putting any constrains on the type of A. On the other hand, as the problem size gets bigger, direct solvers start exhibiting memory problems given their demand for long recurrence. Moreover, and because of the fill in problem, data structure used to store the original sparse coefficients is continuously altered and never preserved as lot of previously zero entries become non zero as the factorization proceeds [18,19].

Over the past 30 years, sparse direct solvers continued to develop and various strategies were introduced to guarantee more stable LU decomposition with minimal fill-in as in that preserves sparsity [20]. Despite all of the attempts, and because of large storage demand and processing requirements not to mention their inherent sequential nature, some authors believe that the use of direct methods in practice is still limited to 2D mathematical modeling [21]. On the other hand, and because of direct methods' superior robustness and because computers are getting faster, others [17] believe many problems will be solved by methods from both approaches.

The most famous direct approach is Gaussian elimination. In its general form, the method decomposes matrix (A) into both lower and upper (LU) triangular forms. To solve a given linear system, forward elimination is performed first before back substitution takes place. With special focus on the sparse case, Scott in [21] considered many numerical examples and reviewed frontal and multifrontal methods that are derived by combining Gauss elimination and finite element approaches. Such methods are characterized by reducing storage and processing demands by interleaving matrix assembly with the elimination steps.

Motivated by Strassen's algorithm [22] that uses recursion to speed up matrix multiplication, not to mention recursion highlighted success in computational problems when applied to dense matrices, Dongarra and others [23] attempted a recursive approach for LU factorization of sparse matrices. Although, they reported efficient storage and speedup compared to multifrontal methods, recursion suffers from substantial drawbacks [24] that limits its scalability and promised performance when implemented on massively parallel machines. First although recursion leads a very concise and readable code, it is sequential in nature as it is executed in memory stack that forces Last In First Out (LIFO) sequence of function calls. Second, recursion demands long recurrences to be computed which in turn require excessive memory storage.

On the other hand, and although they, might be susceptible to divergence issues and compromised accuracy, iterative methods are highly favored in the solution of large sparse systems. First, they preserve the sparsity pattern as they do not attempt modifying the coefficient matrix. Second and most important, beside vector updates, the essential operation in almost all iterative solvers is matrix vector multiplication [18] that is characterized by its inherent parallelism. Moreover, and although iterative approaches are problem specific, it has been shown that the convergence could be enhanced by the use of suitable preconditioners.

Starting with a guess of the unknown vector, and if the solution exists, iterative methods continue to refine the initial solution according to a certain criteria until converging to the desired accuracy. The overall idea lies behind replacing the system of linear equations by some nearby system which is easily solved [19,25]. Iterative solvers could be further classified into two main groups: stationary methods like Jacobi, Gauss Seidel, Successive over Relaxation, and non-stationary like Krylov subspace methods [18,26,27].

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

By definition, the Krylov subspace generated by the coefficient matrix A and the accompanying residual: $r_0 = b - Ax_0$ is denoted by: $K^k(A; r_0)$, with k indicating the iteration and defined as:

$$K^k(A; r_0) \in span\, r_0,\ A\, r_0, A^2 r_0, \ldots, A^{k-1} r_0,\ A^k r_0.$$

According to the way the vector of unknowns (x) is chosen from the constructed subspace that contains the successive approximate solutions, Krylov methods are further classified into different categories as per the following approaches: [19]

1. **The Ritz–Galerkin approach**: constructs x_k for such that the residual is orthogonal to the current subspace: $b - Ax_k \perp K^k(A; r_0)$. This leads to Conjugate Gradients, The Lanczos method, FOM, GENCG methods.
2. **The minimum norm residual approach**: identifies x_k for which the Euclidean norm $\|b - Ax_k\|_2$ is minimal over $K^k(A; r_0)$, then we have: GMRES [28], MINRES, ORTHODIR.
3. **The Petrov–Galerkin approach**: x_k is found so that the residual $b - Ax_k$ is orthogonal to some other suitable k-dimensional subspace. This leads to BiCG [29] and QMR [30].
4. **The minimum norm error approach**: Determine x_k in $A^T K^k(A^T; r_0)$ for which the Euclidean norm $\|x_k - x\|_2$ is minimal. This leads to SYMMLQ and GMERR.
5. **Hybrid Approaches**.
   a. CGS, Bi-CGSTAB [31].
   b. Bi-CGSTAB(L), TFQMR, FGMRES, and GMRESR.

For extensive review of direct solvers and various implementation variations, one might consider [26,32–34]. A comprehensive survey for preconditioning techniques is presented in [35]. For a complete survey on iterative solution methods, please check [17]. For a very quick algorithmic treatment and comparison [26]. The books by [7,19] presents a comprehensive treatment of the subject with a focus on the theory and finally, [36] describes various aspects of the parallel implementation of iterative solvers.

The finite volume discretization of flow equations that governs the behavior of the forward modeling of the two-phase oil–water reservoir will eventually yield a sparse system having ill-conditioned, unsymmetrical coefficient matrix with Hepta-diagonal profile and 2×2 block representing each entry. As mentioned before and despite their tremendous flavors and the dozens of available implementations nicely summarized in [37], picking up a universal and efficient parallel sparse linear solver is very challenging as many mutually interacting factors stand in the way. For instance, the coefficient matrix (A) of our modeled FRS has certain properties that put further restrictions on any solver selection. First, because of its very large dimensions and the resulting sparsity pattern, direct methods will be excluded due to their reported memory demands. Moreover, as (A) is ill-conditioned, stationary iterative methods will not be considered because of issues related to convergence. Finally, since (A) is unsymmetrical, some Krylov based methods dedicated for symmetrical systems will not be taken into account. In addition to the previous constrains and as the intended parallel implementation will eventually serve in either reducing the overall execution time or enabling larger problems to be handled with the same sequential time, the selected solver should have high degree of data independency regardless of the amount of work involved. Furthermore, the selected algorithms should be in harmony with the target parallel architecture, the NVidia's GPU, as the later imposes additional constrains. As a result and before implementing the parallel version of FRS, comprehensive experiments on sample matrices extracted from our sequential FRS implementation were conducted using the parallel CUSP library [38]. The goal was to examine how the parallel execution time of different solvers is affected by different sparse storage mechanisms and to empirically select the one that suites FRS application the most.

2.3. Parallel model for FRS

Just as various complex algorithms and software modeling techniques have emerged as a necessity for developing huge sequential applications, large scale massively parallel programs are in more demand for either making use of such well-established techniques or even developing new aiding tools. This could be attributed to the observed fact that the life cycle of a parallel program is very long, error prone, complex and requires special attention to the underlying hardware resources [39].

As mentioned before, the goal of parallel programming is to provide tools and techniques for either solving big problems faster or to run larger instances of the given problem for the same time interval that was used to execute their serial counterpart. Exposing application concurrency refers to the art of breaking down the main problem into independent logical tasks that could be later executed in parallel after mapping them to corresponding physical processing elements. It is then no wonder that restructuring the problem to exploit any available concurrency is indeed the first mandatory step before implementing any serial algorithm using a suitable parallel programming environment. The process for finding concurrency starts by a decomposition step performed on program data and the associated tasks. It is followed by an analysis step where the decomposed parts are grouped, ordered, or share their own data. Although exposing program concurrency may be achieved by developing and analyzing the dependency graph that in turn may be constructed in many ways [40], such methods are suited to express concurrency of computationally expensive algorithms or small scale systems. As the reservoir simulator application is a little bit more complicated, we tend to utilize more elegant methods from the software engineering general-purpose UML modeling [41,42]. UML provides standard graphs to visualize the design of large scale systems and their associated relations. Throughout the FRS parallel implementation process, several related and complementary diagrams that describe the whole system from various design viewpoints have been constructed to eventually aid in understanding and analyzing the parallel program.

While the Activity Diagram represents the behavioral part of the system, Deployment Diagram, also called Topology or Collaboration Diagram, shows the structural aspect and demonstrates how software and hardware work together [43]. The Deployment Diagram is usually the first recommended step in the modeling of traditional large scale parallel applications [39,44]. The Activity Diagram shows the execution flow of the processes and what actions are performed to achieve an ultimate goal. In the context of parallel application modeling, this diagram provides means of representing communication, synchronization and computational operations [39,42]. Sequence Diagram as well as Communication or Collaboration Diagram, are also utilized to add another perspective to the behavioral description of the system. While the Sequence Diagram depicts dynamic system elements as they interact overtime, Collaboration Diagram also shows how system components are spatially related [45].

2.4. Many-core machines and the CUDA programming model

Many-core machines have emerged naturally as an answer to the continuous demand and need for more performance. They have been developed by considering the tricks and limitations that have been learnt over the years of continuous improvement on the design of both single and multicore systems. In addition to exploiting all possible optimizations to their limits, many-core machines came to existence after realizing that Instruction Level Parallelism (ILP) techniques could only deliver constant factors of speedup [46]. Moreover, it has been firmly realized that clock

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

speed could not be increased anymore without melting the chip. As a result, the design consideration for many-core systems was centered around optimizing the architecture for power rather than performance [47]. On NVidia's GPUs for instance, teraflop performance, or even exaflop in the near future, is achieved via hundreds of thousands cooperating threads performing the same task simultaneously to be later executed on simple cores operating at MHz clock. Unlike the previous trend in manufacturing high performance computing machines, designing dedicated throughput oriented devices rather than utilizing general purpose latency oriented ones had enabled smarter utilization of Moor's observation [48]. Doubling the number of transistors every eighteen month on a chip is now used to create either many-core processors, or single chips having multiple processor cores [46,49,50].

The product line at NVidia is continuously introducing new generations of high performance power efficient hardware. Besides the offered extreme computing capabilities, the new Kepler architecture [51] has introduced more features that enables increased GPU utilization and simplify parallel program design. For example, by allowing kernels to have full control on spawning other kernels, dynamic parallelism gives more flexibility for parallelizing nested loop iterations and performing recursion. Moreover, and to better utilize the system's multicores, Hyper-Q allows multiple simultaneous connection lines from those cores to launch work on the GPU, thus supporting computation and communication overlapping optimization.

With a support to 2688 CUDA Cores, 6 GB memory with 250 GB/s bandwidth, the Tesla K20 GPU is capable of delivering 1.32 TF and 3.95 TF double and single precision peak performance, respectively. The accelerator that is made of more than 7.1 Billion transistors is shipped with 15 streaming multiprocessors (SMX) and 1.5 MB L2 cache. Each SMX supports a maximum of 2048 threads, 16 thread blocks, 64 K 32-bit registers, up to 48 K shared memory. Each thread block can have a maximum of 1024 threads, while every thread can have a maximum of 255 registers. The computing Grid can support a maximum of $2^{32} - 1$ threads. Four warps each containing 32 threads can be issued and executed concurrently. Threads within a warp can share data through the new implemented Shuffle instruction and therefore reduce the amount of shared memory needed per thread block.

The NVidia GPU memory (Fig. 4) is organized at different levels each of which varies in speed, usage, size, and scope [51]. Data stored in global memory are allocated and destroyed from the host and are visible by all threads in the application. With a similar scope and certain considerations, the read only 512 kB Constant Memory provide a relatively faster access speed than the global memory by reducing bandwidth usage through caching constant values and broadcasting them to all threads in a warp. At the block level and being visible to all threads in the block, the configurable shared memory and in the absence of bank conflicts, provide even much faster access speed and allow data sharing and reuse among threads within the block. Finally, and with a lifetime of the thread that created it, registers are considered the fastest memory elements requiring zero clock cycle per instruction in the absence data dependency.

For more detailed information about the architecture, the programming model and suggested optimizations for K20x device please refer to [51–55] for CUDA programming guide.

It is always intimidating to utilize more resources that grant higher throughput like registers, but unfortunately that comes with the price of limiting concurrency. After all, one key aspect through which GPU devices achieve their Tera-flop performance is through latency hiding. When a given warp stalls because of unavailable data and while these data being fetched, other warps are switched and scheduled for execution. Similarly, when a block stalls for any reason, other blocks are switched in by the scheduler. As a result, a smart selection for the number of blocks to be executed as well as the number of threads used by each block is mandatory for any successful exploitation of GPU device capabilities for achieving higher performance.

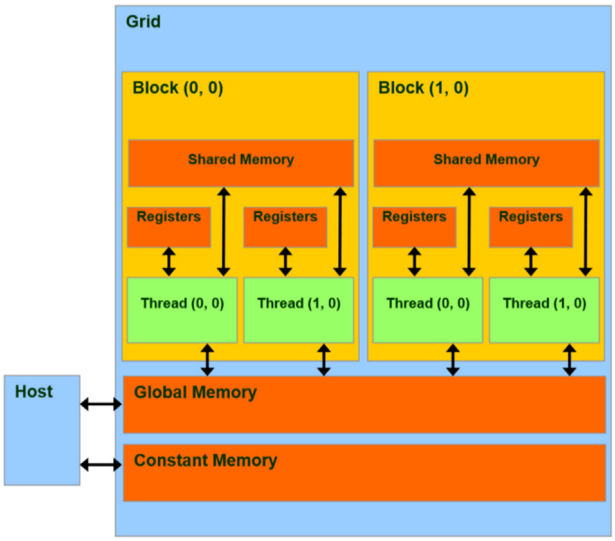

Fig. 4. Memory hierarchy in NVidia GPU.

Each streaming multiprocessor (SMX) in Kepler GK110 supports a maximum of 65 536 registers, 16 blocks, 2048 threads and 64 warps. Forcing CUDA kernel to use registers for variables may be achieved by explicitly using scalar variables and via loop unrolling. However and as mentioned above, using more registers will hinder performance as it limits the number of lunched blocks. For example, assume we are using 256 threads each uses 100 32-bit registers (50 double private variables). Then each block will demand $256 \times 100 = 25\,600$ registers. As a result, the maximum number of blocks that can be launched is calculated by dividing the maximum number of registers supported by each SMX over the utilized registers or $(65\,536/25\,600) \times 2$ blocks. This means utilizing only 12.5% of the maximum blocks allowed per SMX! As a result, a good optimization to be followed lies in decreasing the use of registers as possible and shifting variables to make use of shared memory.

In a similar way and although its latency is almost 100× lower than uncashed global memory latency, the exorbitant use of shared memory may also limit the pledged device performance. If 48 kB of shared memory is to be used among 8 blocks, then each block should utilize a maximum of 6 kB shared memory! Moreover, to prompt for higher bandwidth utilization, shared memory is distributed into concurrently accessed, equally sized 32 4-Bytes logical banks each with bandwidth of 64 bits per clock cycle. Memory bank conflict degrades shared memory performance by serializing bank accesses and occurs when multiple simultaneous requests by different threads are made to the same bank. Therefore, whenever shared memory is utilized, the associated variables should be placed under scrutiny to avoid possible bank conflicts. To enable better optimization when double precision variables are used, device bank size should be configured to be 8 bytes instead of the default on.

NVIDIA provides a CUDA Sparse Matrix library (cuSPARSE) for manipulating and operating on sparse matrices [38]. The library provides a collection of basic linear algebra functions that are called from C++ programs. They reported around 8× faster performance gain over their direct competitor Math Kernel Library (MKL) offered by Intel [56]. The library has been used extensively by the researchers as it provides fast and reliable performance with ease of programming and development effort. For example [57] utilize

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

Table 1
Condition number for various samples of the reservoir simulator.

Matrix size	120,000 × 120,000	Average number of iterations required for convergence with relative tolerance of (1.0e−6)				
Sample time	Condition number	GMRES(5)	GMRES(15)	GMRES(50)	GMRES(1000)	BiCGSTAB
0	1.279E+05	45	39	38	36	26
31	1.112E+06	377	204	155	140	95
62	1.873E+06	484	231	189	169	115
93	3.548E+06	751	434	296	236	164
124	4.708E+06	945	397	378	288	205

it to implement (ILU) and Cholesky factorization for iteratively solving linear systems, while [58] used it to accelerate the modeling of deformation of soft tissue using (FEM). [59] made use of the library to boost image segmentation implementation; and [60] apply it for image reconstruction.

Similar to cuSPARSE, the CUSP library [61] provides a wrapper for many functions in cuSPARSE. It was designed solely to take advantage of the intensive computational aspect of the massively parallel NVidia's GPUs. It is released under the Apache 2.0 open source license. The CUSP library is an inevitable starting point for CUDA developers writing parallel scientific computing applications. The library not only provides abstraction and easy to use call to cuSPARSE and cuBLAS [62] routines, but also reports good performance. Moreover, the developed applications can be smoothly integrated with THRUST library [63] to enable fast prototyping. CUSP could be used directly by including the associated interface files, and provides dozens of graph algorithms and sparse linear algebra routines easily deployed with many available sparse storage schemes and preconditioners.

Simulations that target Oil Reservoir, and just like other scientific applications, have continuously and increasingly attempted to speed up their computations by taking advantage of the excessive throughput offered by the above state of the art massively parallel many-core GPUs. For example, [64–66] presented a parallel CUDA implementation of related preconditioned linear solver, and reported promising speedup after incorporating it with some reservoir simulations. Verified with some predefined collection of preconditioning techniques, Sudan et al. introduced a flexible version of the famous GMRES solver and implemented it on early versions of many-core GPUs using CUDA [67]. They experimented various reordering mechanisms to reduce memory access latency and hence improve the overall execution time. They finally validated the usefulness of the suggested implementation named FGMRES across different practical reservoir simulations. Taking it from there, the authors of [68] have further developed a hybrid novel solver that exploits both data and functional parallelism to orchestrate using some heuristics operations between multicore CPU and many-core GPU. The authors reported what they considered a significant speedup after performing various experiments on complex reservoirs. In their work, [69] attempted to quantify the impact of using the emerging GPUs on reservoir simulation by parameterizing both model size and the number of GPUs as well as utilizing a highly parallel simple linear equation solvers. Their conclusion on the conditionally attained speed up was in accordance with the above studies. They also introduced a development of a special mixed precision solver that provide good performance on four GPU accelerators.

Authors of this paper believe that the promising work of [68] could be further enhanced by utilizing other optimizations that have been made available in current state of art GPU architectures with higher computing capabilities [52,70]. These include: (1) the cautious use of CUDA Pinned memory, page-locked, to exploit Direct Memory Access (DMA) on GPU and enable asynchronous data transfer. This will not only allow better utilization of the PCI-E bus, but also amortize the natural high latency of memory transactions by overlapping computation and communication. (2) Employing some CUDA-Aware MPI implementations [71,72], to directly pass GPU buffers to MPI through the newly introduced Unified Virtual Addressing (UVA) feature. This would result in yet faster code as the underlying transactions could be further pipelined and GPUDirect technologies would be transparently utilized [73]. Also, from the above discussion and despite all the reported promising speed up that relies heavily on the utilized technology, the development process of any parallel reservoir simulator, including the one presented in this paper, is both context and problem dependent because of the many underlying configuration parameters. Hens, we believe that a detailed comparative review for all published and verified parallel reservoir simulators, would be very useful to standardize such a process and to help guide industry to better utilize such cutting edge technologies.

3. Methods

Five large samples at different time iterations of the forward reservoir simulator have been extracted and their condition number was measured, Table 1. Each sample represents a 3-D structured grid with (2 × 2) block entries distributed in a Hepta-diagonal fashion as resulted from finite volume discretization. Sample_0 is assembled at the first time iteration of the simulator, and its coefficients are a combination of various reservoir parameters (permeability, compressibility…), oil pressure values P_o and water saturation levels S_w. When simulation time proceeds, elements composing the coefficient matrix changes as both P_o and S_w get updated. Tests were performed on a node in an HPC cluster offered by the Information Technology Center at KFUPM featuring a Xeon E5-2680 10-Core, 2.8 GHz (Dual-processor) and Tesla k20x GPU [74].

Experiment 1

The first experiment examines how the parallel execution time of different solvers is behaving with different sparse storage mechanisms. Four different sparse storage schemes were considered [26,75]: coordinate storage scheme (COO), Compressed Row Storage (CSR), ELLPACK (ELL) and Hybrid (HYB). The CUSP version of a parallel solver accompanied with the mentioned storage formats for each matrix in Table 1 was tested using different restarted versions of GMRES (5, 15, 50, and 1000) and BiCGSTAB. We made use of the available Bridson approximate inverse preconditioner that reduces the fill-in and improves convergence vi reordering elements in coefficient matrix [76,77]. Each experiment was repeated ten times and the average as well as some statistics were reported.

After getting an idea on the solver that will be utilized, potential parallelism in the whole FRS application was analyzed and a parallel CUDA program for the entire application based on various suggested optimizations was implemented.

Experiment 2

A quick glance at the sequential implementation of the reservoir simulator reveals and in a broader sense a number of write after write [78,79] data hazards for each flow calculation. The issue

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

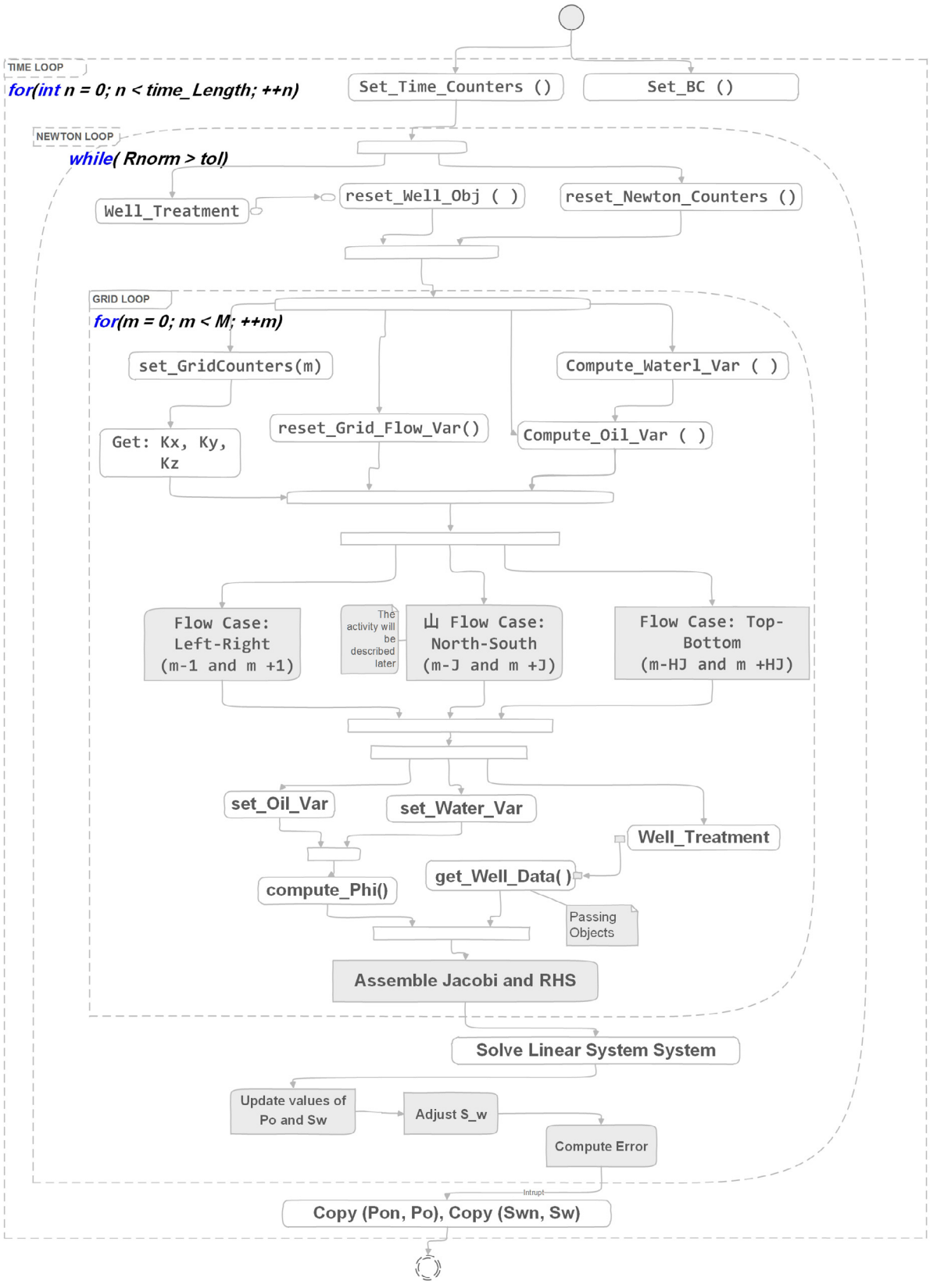

Fig. 5. The activity diagram for the reservoir simulator, with its computational scheme shown in Fig. 3.

has been resolved by giving off some space in order to create independent tasks. Instead of having one variable location being updated sequentially, multiple copies of the same variable have been allocated with proper renaming. Moreover, by refereeing back to FRS Computational Model (Fig. 3), one can establish the associated corresponding detailed Activity Diagram (Fig. 5). Moreover and without loss of generality, the concurrent operations of flow calculations from north to south are shown in Fig. 6.

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

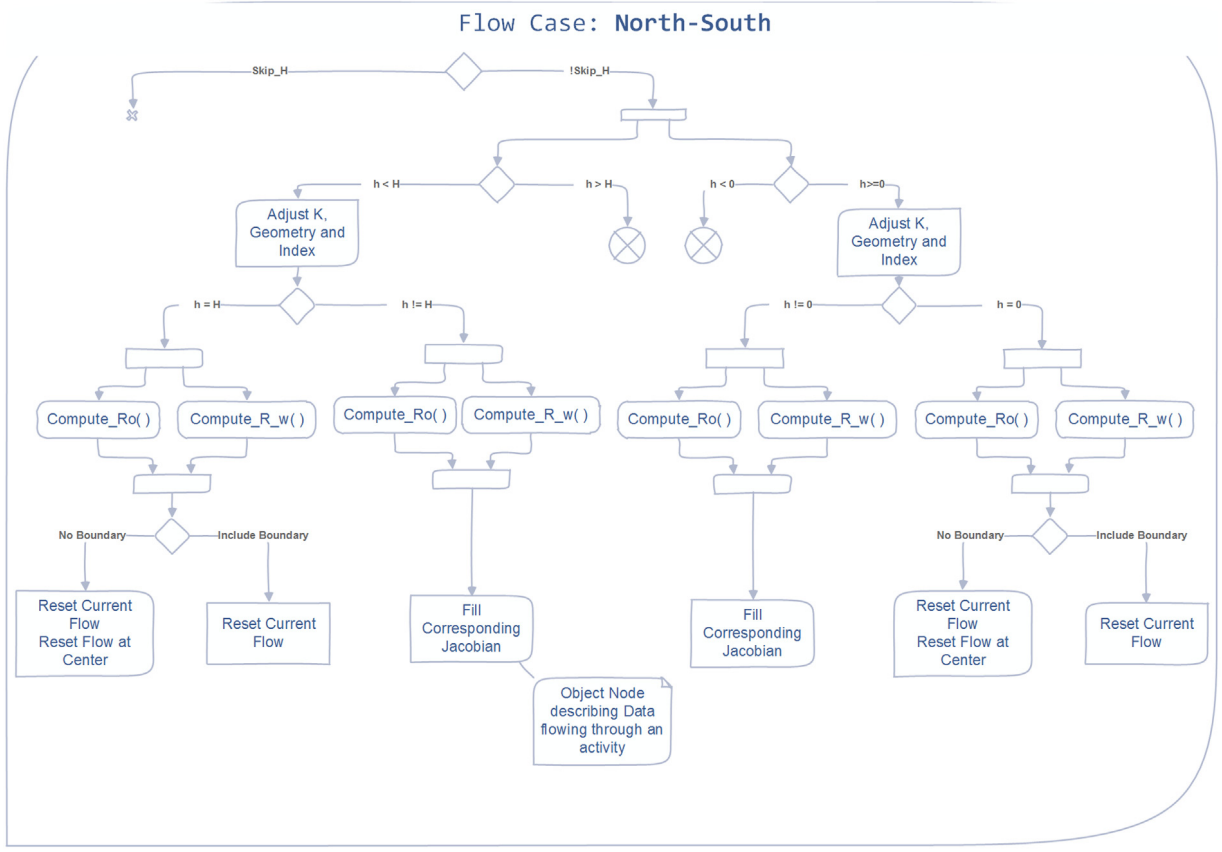

Fig. 6. The activity diagram for a sample North–South flow calculation inside the Newton iteration.

As a result, the following could be concluded about the matrix assembly stage:

- The system operates on large data structures. Basically large arrays that store (P_o and S_w, P_{on} and S_{wn}) values that are shared among all workers.
- The shared data arrays in (P_o and S_w), are solely read during matrix assembly stage. Before passing them to the next iteration they are modified and written back after solving the assembled ill conditioned unsymmetrical sparse linear system.
- Unlike the Newton and time loops, and if managed properly, grid iterations are independent and do not carry dependency.
- Data portions of the arrays are read independently, for every flow direction.
- The update of the variables inside the array is done through multiple consecutive function calls.

The previous behavior and the established notes suggest that the parallelization process starts by data decomposition step over the large arrays and incorporate task decomposition whenever needed.

The process of data decomposition is about mapping a global index space into a task local index space [47]. It is associated with a granularity level that determines the amount of data each chunk holds. The more the granularity gets smaller, the more independent tasks are created and the more communication overhead to manage the dependencies among chunks is required. It has been suggested that a good data decomposition will poses the following characteristics [47]:

- It has to yield dependencies that scale at a lower dimension than the computational effort associated with each chunk; i.e. making chunks large enough so that the computational effort required to update data, offsets any resulted dependency overhead. Moreover, larger chunks will offer more flexibility when scheduling operations on the processors.
- Preserves load balancing among the execution elements. If not, then the speed at which the computation finishes will be haunted by the speed of the lowest process; i.e. the one with more work. This will be soon reflected on the overall performance as the problem being parallelized is scaled via either increasing resources or problem dimension. After all, better scaling is achieved through the minimization data movement and reducing the serial bottlenecks like barriers to the limit [46].

The analyzed concurrency pattern presents an additional force that influences the way tasks are mapped to processing elements. The simulator consists of multiple independent tasks or weakly related tasks that share a common data structure as well as a sequence of tasks with a static and regular flow ordering pattern. When applicable the so called not true dependency was removed by suitable code transformations. Moreover, a replication of the data structure was done when necessary. Whenever applicable, the whole program has been restructured to create more work with more potential concurrency. Also, optimized routines in Thrust library like reduction and their special data structure has been employed and utilized.

Throughout the program execution, each function call can be thought of as a task which in turn may be composed of other tasks. Moreover, as the iterations in the most inner loop that spans all grid points are independent, each iteration, or even group of iterations, could be thought of as a separate task that in turns operate on its assigned data portion. Again, the general rule of thumb lies in ensuring the creation of enough independent tasks that keep processors busy. In CUDA terms, a global function will launch enough number of thread blocks that handles specific portion of the input data. Threads in the associated blocks will then bring to shared or local memory necessary related data, calling any necessary device functions and operate on them. Table 2 lists some

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

Condition Number		1.279E+05				Matrix_Sample		0			Conf. Coeff: 1.96							
Solver	Sparcity	Parallel Execution Time										Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10								
GMRES(5)	HYP	1.75	1.73	1.74	1.73	1.77	1.74	1.77	1.74	1.77	1.74	1.748	0.016	0.010	1.758	1.738	1.770	1.730	0.040
	ELL	1.74	1.73	1.77	1.75	1.77	1.74	1.76	1.73	1.74	1.73	1.746	0.016	0.010	1.756	1.736	1.770	1.730	0.040
	CSR	1.63	1.62	1.7	1.62	1.6	1.59	1.59	1.62	1.59	1.59	1.615	0.034	0.021	1.636	1.594	1.700	1.590	0.110
	COO	1.69	1.69	1.73	1.69	1.74	1.69	1.75	1.68	1.75	1.72	1.713	0.028	0.017	1.730	1.696	1.750	1.680	0.070
GMRES(15)	HYP	1.83	1.79	1.73	1.72	1.72	1.72	1.72	1.74	1.71	1.74	1.742	0.038	0.024	1.766	1.718	1.830	1.710	0.120
	ELL	1.75	1.73	1.72	1.74	1.76	1.75	1.75	1.75	1.75	1.74	1.744	0.012	0.007	1.751	1.737	1.760	1.720	0.040
	CSR	1.64	1.62	1.63	1.62	1.65	1.64	1.63	1.63	1.62	1.62	1.630	0.011	0.007	1.637	1.623	1.650	1.620	0.030
	COO	1.67	1.71	1.68	1.72	1.68	1.68	1.67	1.66	1.63	1.66	1.676	0.025	0.016	1.692	1.660	1.720	1.630	0.090
GMRES(50)	HYP	1.8	1.77	1.75	1.75	1.77	1.76	1.74	1.76	1.78	1.78	1.766	0.018	0.011	1.777	1.755	1.800	1.740	0.060
	ELL	1.77	1.76	1.76	1.77	1.75	1.76	1.78	1.8	1.75	1.75	1.765	0.016	0.010	1.775	1.755	1.800	1.750	0.050
	CSR	1.65	1.69	1.65	1.66	1.65	1.66	1.64	1.65	1.7	1.66	1.661	0.019	0.012	1.673	1.649	1.700	1.640	0.060
	COO	1.75	1.69	1.67	1.69	1.68	1.67	1.68	1.72	1.7	1.76	1.701	0.032	0.020	1.721	1.681	1.760	1.670	0.090
GMRES(1000	HYP	1.94	1.9	1.88	1.9	1.94	1.89	1.89	1.94	1.89	1.89	1.906	0.024	0.015	1.921	1.891	1.940	1.880	0.060
	ELL	1.89	1.91	1.87	1.88	1.92	1.88	1.91	1.88	1.89	1.88	1.891	0.017	0.010	1.901	1.881	1.920	1.870	0.050
	CSR	1.77	1.78	1.77	1.78	1.8	1.77	1.78	1.79	1.77	1.79	1.780	0.011	0.007	1.787	1.773	1.800	1.770	0.030
	COO	1.81	1.82	1.81	1.85	1.82	1.84	1.82	1.84	1.8	1.81	1.822	0.016	0.010	1.832	1.812	1.850	1.800	0.050
BiCGSTAB	HYP	1.87	1.81	1.81	1.85	1.81	1.86	1.81	1.8	1.85	1.81	1.828	0.026	0.016	1.844	1.812	1.870	1.800	0.070
	ELL	1.78	1.76	1.78	1.7	1.71	1.75	1.71	1.73	1.72	1.71	1.735	0.030	0.019	1.754	1.716	1.780	1.700	0.080
	CSR	1.69	1.64	1.72	1.7	1.66	1.66	1.67	1.71	1.67	1.71	1.683	0.027	0.017	1.700	1.666	1.720	1.640	0.080
	COO	1.73	1.73	1.73	1.72	1.73	1.77	1.74	1.84	1.74	1.76	1.749	0.035	0.022	1.771	1.727	1.840	1.720	0.120

Fig. 7. Average parallel execution times for Sample_0.

Table 2
List of utilized optimizations in the developed parallel FRS code.

Target optimizations	Details
Shared memory utilization	Intensive use of device shared memory and making use of its broadcast property to serve data among threads at a fast pace.
Titling	To handle large vectors, each thread at first load data into shared memory and performs the corresponding desired operation. It then stores the result back to global memory before another kernel take data accumulated in this new vector in global memory and continue operating on it.
Memory coalescing	A *warp* can access a number of successive memory locations in a single transaction. Therefore, maximizing BW utilization.
Occupancy and latency hiding	Launching enough threads to keep resources busy.
Data transfer	Minimize copying, and makes use of asynchronous data transfer between host and device by utilizing pinned memory and streams. Kepler GK110 introduces HyperQ mechanism that supports 32 hardware managed connections for communication between host and device. As a result, device utilization has been increased as multiple processors on the CPU could initiate work on a single GPU at the same time.
Overlap communication and computation	Host and kernel execution overlap: when possible, the original code was restructured in a way that a call to device kernel is followed by a many calls to host functions. By default, kernel launch is asynchronous or non-blocking. So while the GPU is busy, the host computes part of the algorithm. If used properly, this mix, combined with streaming has great impact on performance.
Computation intensity	Loop unrolling was utilized to further increase computation intensity.

utilized optimizations in the developed FRS code. More detailed information with examples could be found in [80,81].

Unlike the IMPES method, our developed simulator utilizes an implicit formulation which does not impose any restriction on the time-step. However, to maintain an acceptable level of accuracy, the time-step should not be too large particularly during the early times (the start of simulation). Nonlinear convergence is checked as follows: first, we find the l_2-norm of the error in each phase and compare the highest residual of all phases with a preset tolerance (1e−6). Before making the comparison, we multiply each residual by Δt and divide by bulk volume of each grid block (in bbl). I.e. inside each grid iteration we calculate the following residuals:

$$R_{wn} = \frac{R_{wn}}{R_{wn} + R_w^2} \qquad R_{on} = R_{on} + R_o^2$$

and before going to the next time step, the following is calculated:

$$R_{w\_norm} = (\Delta t / V_b)\sqrt{R_{wn}}, \qquad R_{o\_norm} = (\Delta t / V_b)\sqrt{R_{on}},$$
$$R_{norm} = \max(R_{w\_norm}, R_{o\_norm})$$

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

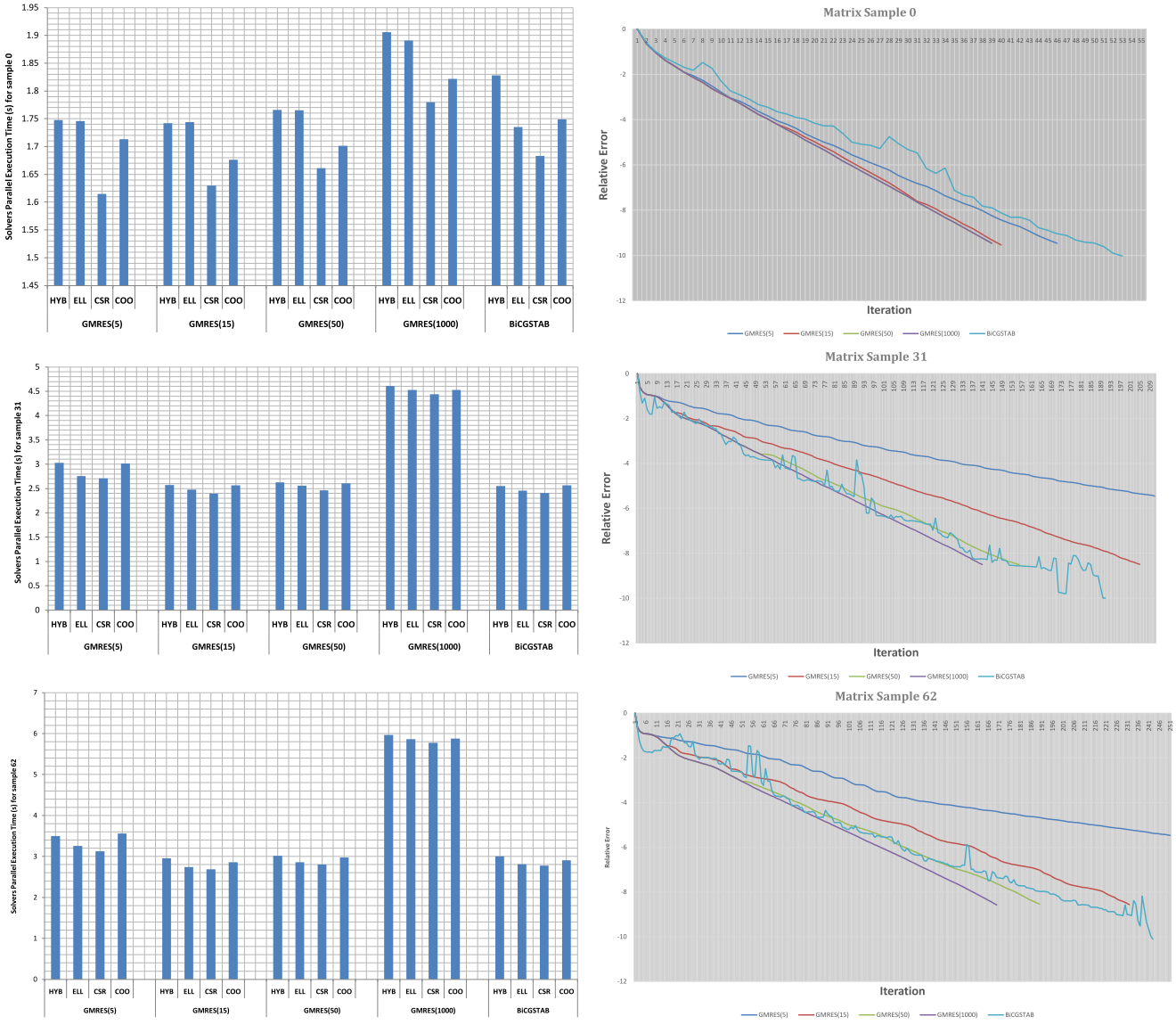

Fig. 8. The average execution time schemes for parallel implementation of BiCGSTAB and GMRES(m) solver using CUSP for different storage. Measurements were separately taken for each sample matrix that was extracted from our implemented FRS (Table 1).

Table 3
Various constants utilized in the developed FRS.

Model properties	Values	
Rock compressibility	5e−6	
Isothermal compressibility	*Water* *Oil*	5e−7 1.2e−5
Initial reservoir porosity, assumed uniform	0.25	
Constant for the equation used when computing viscosity	$Water(c_{\mu w})$ $Oil(c_{\mu o})$	6e−8 2e−6
Initial density	$Water(\rho_{w\_sc})$ $Oil(\rho_{o\_sc})$	62.238 40
Skin factor	1.2	
Well radius	0.25	

The reason for scaling with $\Delta t/V_b$ is to ensure a uniform tolerance for different problem sizes and different time-steps. That is, regardless of the time-step used or grid dimensions used, the same tolerance can be utilized.

Table 3 shows some utilized values that describe our sample model.

The previous described model was implemented and the obtained execution time was compared with a serial version that makes use of Eigen library [82]. The fact that the permeability tensor $\underline{k}$ is nonsmooth results in the condition number of the Jacobian matrix being high. Correctness of results has been verified by comparing the output pressure values from the two programs

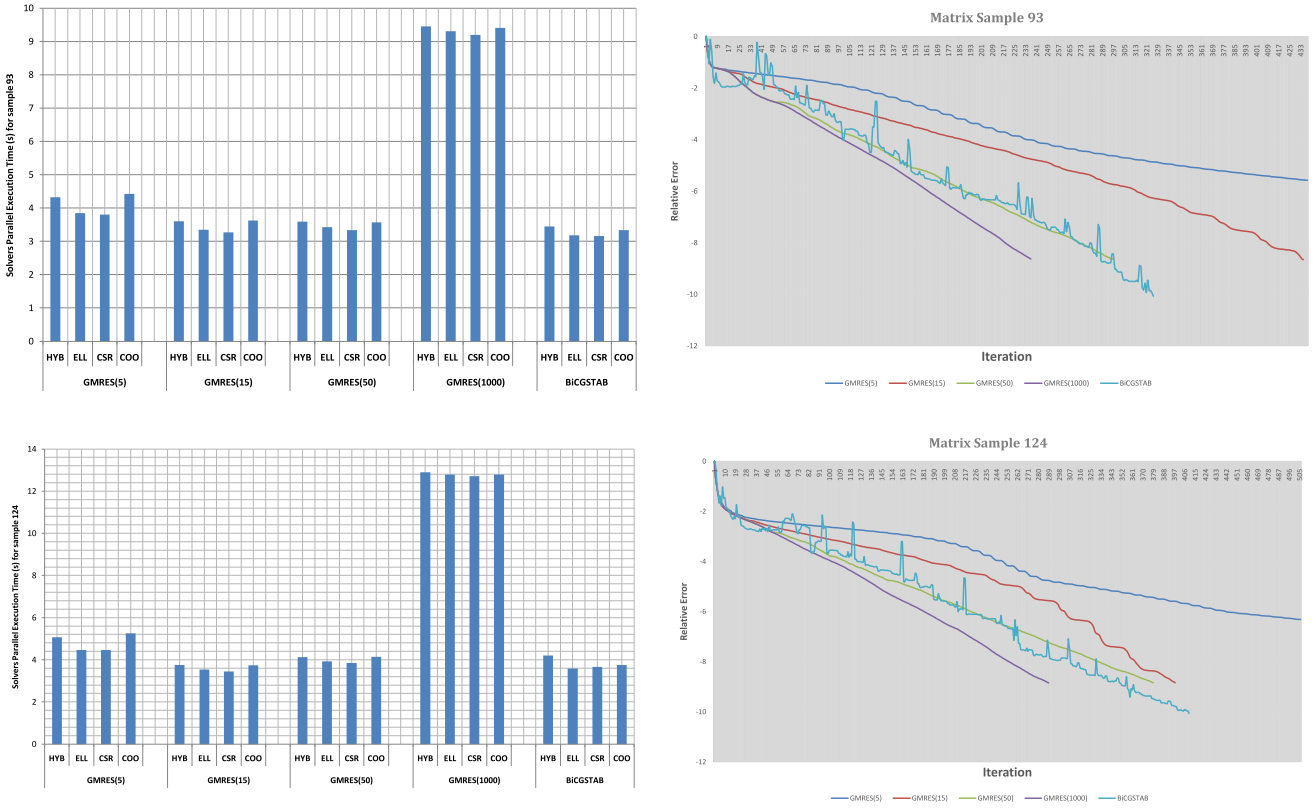

Fig. 8. (*continued*)

Table 4
Well distribution for both the producer and the injector over grid space of $(20 \times 30 \times 2)$.

X-Coor	Y-Coor	Z-Coor	Stb/day	P limit (Psi)
1	1	1	−550	7000
10	1	1	−850	7000
5	5	1	550	2000
1	10	1	350	2000
10	10	1	600	2000
1	20	1	−550	7000
10	20	1	−850	7000
5	15	1	500	2000
15	5	1	600	2000
20	1	1	−550	7000
20	10	1	650	2000
15	15	1	600	2000
20	20	1	−550	7000

for the given well distribution (Table 4). Source code along with detailed comments that relate to the above optimizations will be documented in a separate work.

4. Results and discussion

Without loss of generality, Fig. 7 shows the obtained results of running experiment 1 on sample 0, along with some useful statistics. Fig. 8 shows the results of plotting the execution time for different matrix storage schemes at different samples drawn from our simulator and for the two mentioned preconditioned iterative linear solvers. Every Sample plot is accompanied with another semi-log plot that shows the relative residual per-iteration with a minimum[2] tolerance value of 1e−6. Let i be the iteration number, and the residual $Res = \|r - Ax\|_2$, then the relative-residual is calculated as $Rel_{Res} = \log_{10}(res\,(i)\,/res\,(0))$.

The following could be concluded:

- As time step in our reservoir advances, more iterations would be needed for reaching an accepted convergence level. This is clearly seen in the relative error plot as it is steeper in early reservoir samples. Compare for instance the relative error in Sample_0 and Sample 93. The previous behavior is due to an increase in the condition number of the assembled system as the time advances; the thing that in turns require more iteration to converge.
- A proper restarted version of $GMRES(m)$ may be shown to outperform BiCGSTAB for different storage formats. However, automatic identification of an optimal restart value is not possible. Moreover, and although $GMRES(m)$ enjoys a smoother convergence behavior shown in the relative residual plot, it demands lot of storage space.
- Even for the same matrix structure but with different element values, it is difficult to specify an absolute storage scheme that gives the best performance time. For example, in Sample_0 and for all solvers, COO outperforms HYB. This is not the case for *GMRES* (5) in Sample_93. This could be attributed to the utilized preconditioner that approximate the inverse by exploiting the reordering property to minimize the fill in [61,76,77]. Although, various other assumptions related to implementation considerations, randomness of block execution or matrices condition number could be established as valid vindications, further investigations should be carried out to give more reasonable justification.

Fig. 9 shows the execution time of BiCGSTAB Solver for all considered samples with different storage schemes. The following could be further established and emphasized:

- Different Hepta-matrix data layout has different impact on the solver convergence time. Hence, the solver may have different execution time depending on the simulation step and its corresponding data layout.

[2] It may also reach 1e−7 or 1e−8 depending on the matrix sample.

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

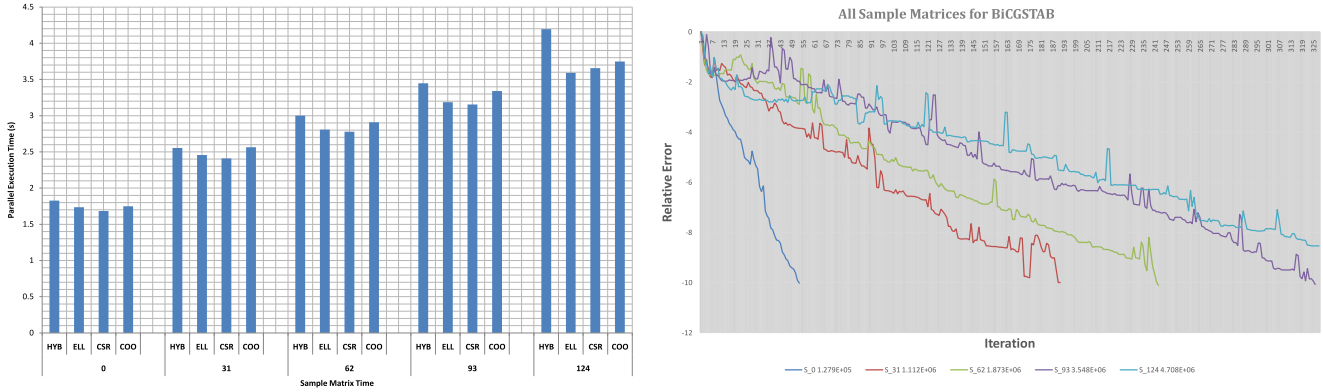

Fig. 9. The average execution time for parallel implementation of BiCGSTAB using CUSP for all matrix samples and different storage schemes. Measurements were performed on various test matrices extracted from our implemented FRS. Table 1.

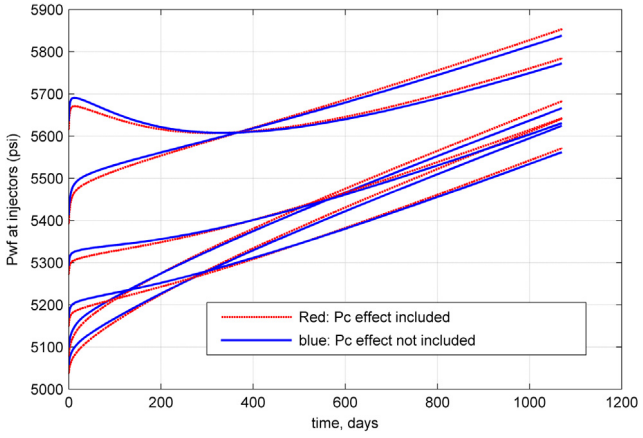

Fig. 10. Pwf at injectors, Pc is included, no flow BC for $20 * 30 * 2$, specified flow rate at injector.

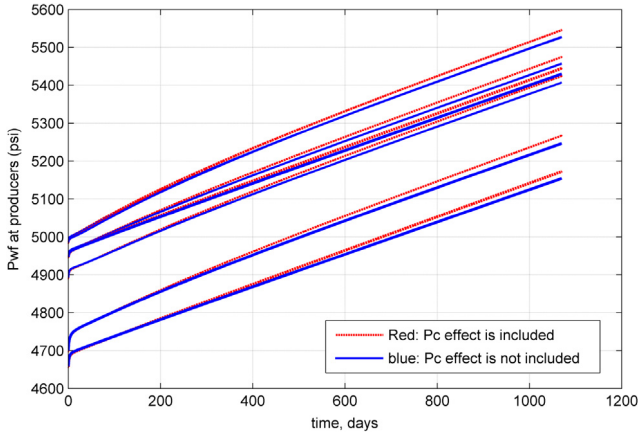

Fig. 11. Pwf at producers, Pc is included, no flow BC for $20 * 30 * 2$, specified total rate at producer.

- The larger the matrix sample, the higher the condition number and the longer it takes to converge.
- BiCGSTAB with CSR storage scheme outperformed others from Samples_0 to Sample_93. It came second in Sample_124. As a result it seems a reasonable choice for our implementation.

Validating the correctness of parallel program output was done in two stages. First, an already verified MATLAB code developed by Abeeb in [5,83] was compared against the implemented serial C++ program for small grid dimensions ($20 \times 30 \times 2$). No flow boundary condition was initially assumed, six injectors with

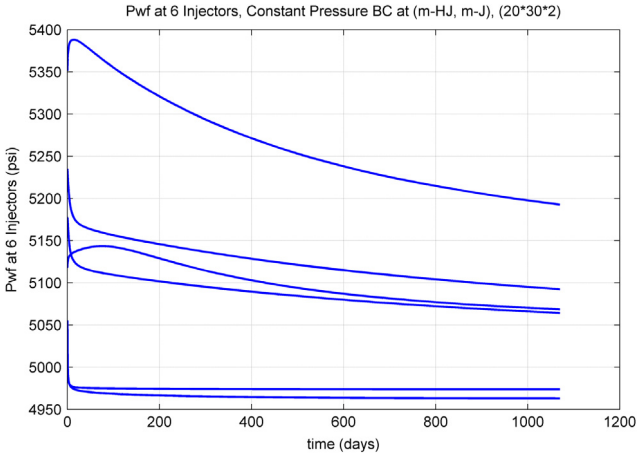

Fig. 12. Pwf at injectors. Constant flow BC (5000psi) at m-J and m-HJ, no flow BC for the rest. Water–oil reservoir of dimensions $(20 * 30 * 2)$ and specified flow rate at 6 injectors.

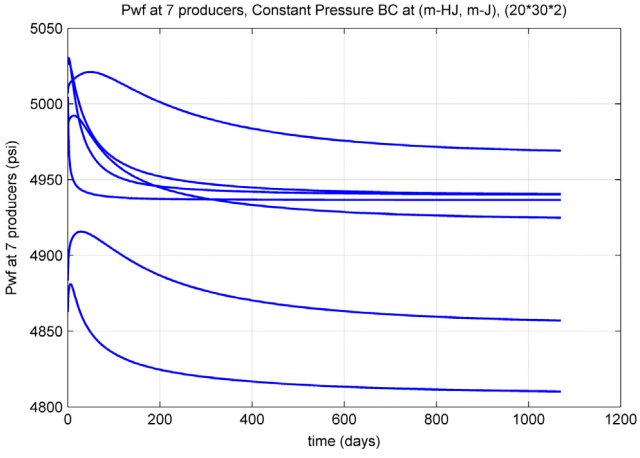

Fig. 13. Pwf at producers. Constant flow BC (5000psi) at m-J and m-HJ, no flow BC for the rest. Water–oil reservoir of dimensions $(20 * 30 * 2)$ and specified total rate at 7 producers.

specified water rate and seven producers with specified total rate were utilized. Fig. 14 shows the permeability map with the distribution of wells taken from Table 4 shown on the map.

Fig. 10 demonstrates the verified output when the effect of capillary pressure (P_c) is included by plotting the flowing bottom-hole pressure (P_{wf}) in psi for the injectors and producers while Fig. 11 plots the results of a similar configuration where capillary was not included. Next Figs. 12 and 13 show the running

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

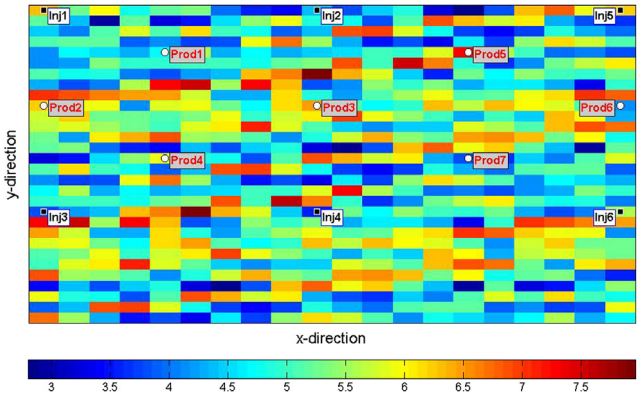

Fig. 14. Permeability map for the utilized wells shown in Table 4.

Reservoir Dimension	X	Y	Z
	240	240	2
Coefficient Matrix Leading Dimension	230400		
Average Serial Execution Time (sec)	18693.25		
Average Parallel Execution Time (sec)	570.07		
Speed Up	33		

Fig. 15. The execution time (ET) for serial and parallel FRS.

simulator when constant pressure boundary condition is applied from certain directions (m-HJ, m-J) with a value of 5000 psi, no flow boundary condition was assumed for all other directions. Again six injectors with specified water rate and seven producers with specified total rate were utilized.

In the second verification phase, larger grid dimensions ($240 \times 240 \times 2$) were considered and tested for both the serial C++ code and the developed parallel version. As mentioned before, the serial version uses Eigen library to provide implementation of the BiCGSTAB solver and the ILU preconditioner while the parallel program calls our developed parallel routine for BiCGSTAB based on various cuSPARSE and cuBLAS library calls. Similar to the work in [57] the parallel version of the ILU preconditioner available in NVidia cuSPARSE Library was also utilized to speed up convergence. The output of pressure and water-cut values for both versions (the serial and the parallel code) were the same. Fig. 15 shows the execution time and the obtained speedup.

Next Figs. 16 and 17 demonstrate how the parallel execution time of the entire FRS varies when doubling reservoir dimension. The objective is two folded: First, to quantify the importance of the above obtained speedup shown in Fig. 15 and see what reservoir dimension is simulated in the same time used to produce results in the serial version. Second: to get an idea on how the developed parallel FRS scales when increasing problem size so that further optimizations could be implemented in subsequent work. For the sake of experimentations, only 25 wells were used.

In accordance with common observation on GPUs, data shows that the GPU simulation becomes more efficient with increasing model size. They reflect the fact that GPUs need a large amount of independent work to operate at maximum efficiency. The serial implementation of FRS took 311.55 min to solve a problem with 230,400 grids. On the other hand, the interpolated data from Fig. 16, speculates that a problem with 18,873,402 grids could be solved in parallel in 311.55 min. In other words the CUDA parallel implementation of FRS enables solving an 82 times larger grid dimension, given the same time to produce results from the counterpart serial implementation.

5. Conclusion

This work has presented a CUDA based parallel implementation for a flexible, two phase, 3D Forward Reservoir Simulation (FRS). Results showed that CUDA parallel implementation of FRS enables solving an 82 times larger problem than the serial counterpart. Moreover, if accompanied by proper preconditioning, BiCGSTAB was shown to be a stable solver that could be incorporated in such simulations instead of the more expensive and usually utilized GMRES that demands storage because of long recurrences. Despite the achieved performance, current implementation uses many registers per kernel, the thing that restricts block concurrency and affects thread latency hiding. Various optimization opportunities, detailed documentation of the implementation as well as the source code will be described in a separate work. Besides the mentioned observations that required more in depth investigation,

Parallel Oil Reservoir Simulation												Conf. Coeff: 1.96							
Coeff. Mat. Leading Dim.	Parallel Execution Time(sec)											Average	STD	Margin Error	Upper Bound	Lower Bound	Max	Min	Range
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10									
10800	75.0	74.8	74.6	74.5	74.7	74.7	74.8	74.7	74.7	74.7		74.72	0.12	0.07	74.80	74.65	74.96	74.50	0.46
28800	170.6	170.5	170.6	170.9	170.6	170.9	170.6	170.2	170.6	170.6		170.61	0.19	0.12	170.72	170.49	170.89	170.23	0.66
57600	259.4	258.6	258.9	258.9	258.7	259.0	258.7	258.7	259.0	259.0		258.89	0.24	0.15	259.04	258.74	259.41	258.57	0.84
115200	414.8	414.5	415.0	414.7	414.6	414.2	414.6	415.1	414.9	414.2		414.66	0.32	0.20	414.85	414.46	415.10	414.16	0.94
230400	570.3	570.2	569.5	570.7	570.0	569.6	570.9	570.1	569.9	569.6		570.07	0.47	0.29	570.35	569.78	570.92	569.50	1.42
460800	885.8	884.4	884.0	884.4	885.0	885.5	885.2	884.0	885.5	885.3		884.91	0.67	0.42	885.32	884.49	885.82	883.97	1.85
921600	1162.3	1165.9	1164.8	1165.9	1164.3	1165.5	1165.4	1164.2	1164.0	1165.2		1164.75	1.09	0.68	1165.43	1164.07	1165.93	1162.33	3.60
1843200	1879.6	1882.8	1881.5	1881.6	1882.5	1877.9	1881.9	1882.6	1881.0	1880.1		1881.14	1.53	0.95	1882.09	1880.19	1882.80	1877.94	4.86

Fig. 16. The parallel execution time of CUDA based FRS for various grid dimensions.

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

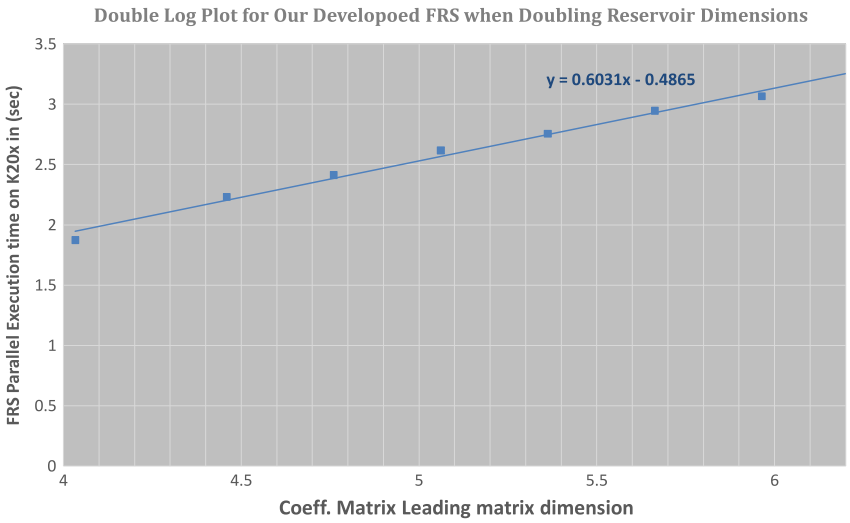

Fig. 17. A double-log plot for the parallel execution time of our developed FRS for various geometries.

implementing a parallel oil reservoir in CUDA is only the first step for many interesting studies to come. Future work includes: FRS based MIC implementation, FRS based OpenACC implementation, FRS on a cluster of GPUs, utilizing Multigrid preconditioners, testing different variants of Krylov solvers and others.

Acknowledgments

We are grateful for the facilities and support provided by KFUPM and the IT center. Thanks also to editors and the anonymous reviewers for their very perceptive and insightful comments.

References

[1] A. Thompson, G.R. Bowen, Parallelisation of an oil reservoir simulation, in: H. Liddell, A. Colbrook, B. Hertzberger, P. Sloot (Eds.), High-Performance Computing and Networking, Vol. 1067, Ed: Springer Berlin, Heidelberg, 1996, pp. 20–28.
[2] J. Abou-Kassem, S.M.F. Ali, M.R. Islam, Petroleum Reservoir Simulation a Basic Approach, Gulf Pub. Co., Houston, TX, 2006.
[3] Z.E. Heinemann, Fluid flow in porous media, 2005. Available: http://edces.netne.net/files/HEINEM~1.PDF.
[4] I.V. Minin, O.V. Minin, Computational Fluid Dynamics Technologies and Applications [S.l.], INTECH, 2011.
[5] A.A. Awotunde, Relating time series in data to spatial variation in the reservoir using wavelets (Ph.D. Thesis), Department of Energy Resource Engineering, Stanford University, 2010.
[6] R. Shahnaz, A. Usman, I.R. Chughtai, Review of storage techniques for sparse matrices, in: 9th International Multitopic Conference, IEEE INMIC 2005, 2005, pp. 1–7.
[7] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, 2003.
[8] R.W. Vuduc, Automatic Performance Tuning of Sparse Matrix Kernels, Citeseer, 2003.
[9] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, N. Koziris, Performance evaluation of the sparse matrix–vector multiplication on modern architectures, J. Supercomput. 50 (2009) 36–77.
[10] S. Xu, W. Xue, H.X. Lin, Performance modeling and optimization of sparse matrix–vector multiplication on NVIDIA CUDA platform, J. Supercomput. 63 (2013) 710–721.
[11] V. Karakasis, G. Goumas, N. Koziris, A comparative study of blocking storage methods for sparse matrices on multicore architectures, in: Computational Science and Engineering, 2009, CSE'09. International Conference on, 2009, pp. 247–256.
[12] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, L. Rao, Optimizing SpMV for diagonal sparse matrices on GPU, in: Parallel Processing, ICPP, 2011 International Conference on, 2011, pp. 492–501.
[13] L. Yuan, Y. Zhang, X. Sun, T. Wang, Optimizing sparse matrix vector multiplication using diagonal storage matrix format, in: High Performance Computing and Communications, HPCC, 2010 12th IEEE International Conference on, 2010, pp. 585–590.
[14] P. Stathis, S. Vassiliadis, S. Cotofana, A hierarchical sparse matrix storage format for vector processors, in: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 2003, p. 8.
[15] P. Stathis, S. Cotofana, S. Vassiliadis, Sparse matrix vector multiplication evaluation using the BBCS scheme, in: in Proc. of 8th Panhellenic Conference on Informatics, 2001.
[16] J. Godwin, J. Holewinski, P. Sadayappan, High-performance sparse matrix–vector multiplication on GPUs for structured grid computations, in: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, 2012, pp. 47–56.
[17] Y. Saad, H.A. van der Vorst, Iterative solution of linear systems in the 20th century, J. Comput. Appl. Math. 123 (2000) 1–33.
[18] R. Mehmood, J. Crowcroft, Parallel iterative solution method for large sparse linear equation systems, University of Cambridge, Computer Laboratory 650, 2005.
[19] H.A. Van der Vorst, Iterative Krylov Methods for Large Linear Systems Vol. 13, Cambridge University Press, 2003.
[20] H.M. Markowitz, The elimination form of the inverse and its application to linear programming, Manage. Sci. 3 (1957) 255–269.
[21] J. Scott, Sparse direct methods: An introduction, in: Electronic Structure and Physical Properies of Solids, Ed: Springer, 2000, pp. 401–415.
[22] V. Strassen, Gaussian elimination is not optimal, Numer. Math. 13 (1969) 354–356.
[23] J. Dongarra, V. Eijkhout, P. Luszczek, Recursive approach in sparse matrix LU factorization, Sci. Program. 9 (2001) 51–60.
[24] A. Kamthane, Programming in C, 2/e, Pearson Education India, 2011.
[25] D.S. Watkins, Fundamentals of Matrix Computations, Vol. 64, John Wiley & Sons, 2004.
[26] R. Barrett, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, SIAM, Philadelphia, 1994.
[27] G.H. Golub, C.F. Van Loan, Matrix Computations, Vol. 3, JHU Press, 2012.
[28] Y. Saad, M.H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986) 856–869.
[29] R. Fletcher, Conjugate gradient methods for indefinite systems, in: Numerical Analysis, Ed: Springer, 1976, pp. 73–89.
[30] R.W. Freund, N.M. Nachtigal, QMR: a quasi-minimal residual method for non-Hermitian linear systems, Numer. Math. 60 (1991) 315–339.
[31] H.A. Van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (1992) 631–644.
[32] I.S. Duff, A.M. Erisman, J.K. Reid, Direct Methods for Sparse Matrices, Clarendon Press Oxford, 1986.
[33] T.A. Davis, Direct Methods for Sparse Linear Systems, SIAM, 2006.
[34] A. Gupta, Recent advances in direct methods for solving unsymmetric sparse systems of linear equations, ACM Trans. Math. Software (TOMS) 28 (2002) 301–324.
[35] M. Benzi, Preconditioning techniques for large linear systems: a survey, J. Comput. Phys. 182 (2002) 418–477.
[36] J.M. Bahi, S. Contassot-Vivier, R. Couturier, Parallel Iterative Algorithms: From Sequential to Grid Computing, CRC Press, 2007.
[37] J. Dongarra, Freely available software for linear algebra on the web, 2000, April, 2003. URL: http://www.netlib.org/utk/people/JackDongarra/la-sw.html.
[38] C. Nvidia, cuSPARSE library, NVIDIA Corporation, Santa Clara, California, 2011.
[39] Y. Perez-Riverol, R.V. Alvarez, A UML-based approach to design parallel and distributed applications, 2013, arXiv Preprint arXiv:1311.7011.
[40] F. Gebali, Algorithms and Parallel Computing, Vol. 84, John Wiley & Sons, 2011.
[41] S. Pllana, T. Fahringer, Modeling parallel applications with UML, in: 15th International Conference on Parallel and Distributed Computing Systems, PDCS 2002, 2002, pp. 19–21.

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010

[42] H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, in: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 737–738.
[43] J.B. Warmer, A.G. Kleppe, The Object Constraint Language: Precise Modeling With Uml, in: Addison-Wesley Object Technology Series, 1998.
[44] S. Pllana, T. Fahringer, UML based modeling of performance oriented parallel and distributed applications, in: Simulation Conference, 2002. Proceedings of the Winter, 2002, pp. 497–505.
[45] S.S. Alhir, Learning Uml, O'Reilly Media, Inc., 2003.
[46] M. McCool, J. Reinders, A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Elsevier, 2012.
[47] T.G. Mattson, B.A. Sanders, B.L. Massingill, Patterns for Parallel Programming, Pearson Education, 2004.
[48] R.W. Shonkwiler, L. Lefton, An Introduction to Parallel and Vector Scientific Computation, Vol. 41, Cambridge University Press, 2006.
[49] C. Campbell, A. Miller, A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures, Microsoft Press, 2011.
[50] S.U. Khan, L. Wang, A.Y. Zomaya, Scalable Computing and Communications: Theory and Practice, 2013.
[51] NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012, Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-ArchitectureWhitepaper.pdf.
[52] N. Wilt, The Cuda Handbook: A Comprehensive Guide to GPU Programming, Pearson Education, 2013.
[53] C. Nvidia, Programming guide, ed, 2008.
[54] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.
[55] D.B. Kirk, W.H. Wen-mei, Programming Massively Parallel Processors: A Hands-On Approach, Newnes, 2012.
[56] M. Intel, Intel Math Kernel Library, ed, 2007.
[57] M. Naumov, Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS, Nvidia White Paper, 2011.
[58] B. Yang, X. Chen, X.Y. Liao, M.L. Zheng, Z.Y. Yuan, FEM-based modeling and deformation of soft tissue accelerated by cuSPARSE and cuBLAS, Adv. Mater. Res. 671 (2013) 3200–3203.
[59] K. He, S.X.-D. Tan, E. Tlelo-Cuautle, H. Wang, H. Tang, A new segmentation-based GPU-accelerated sparse matrix–vector multiplication, in: Circuits and Systems, MWSCAS, 2014 IEEE 57th International Midwest Symposium on, 2014, pp. 1013–1016.
[60] L.A. Flores, V. Vidal, P. Mayo, F. Rodenas, G. Verdú, CT image reconstruction based on GPUs, Procedia Comput. Sci. 18 (2013) 1412–1420.
[61] N. Bell, M. Garland, Cusp library, 2012, ed.
[62] C. Nvidia, Cublas library, NVIDIA Corporation, Santa Clara, California, Vol. 15, 2008.
[63] J. Hoberock, N. Bell, Thrust: A parallel template library, vol. 42, p. 43, 2010. Online at http://thrust.googlecode.com.
[64] J. Appleyard, J. Appleyard, M. Wakefield, A. Desitter, Accelerating reservoir simulators using GPU technology, in: SPE Reservoir Simulation Symposium, 2011.
[65] Z. Chen, H. Liu, S. Yu, Development of algebraic multigrid solvers using GPUs, 2013.
[66] S. Yu, H. Liu, Z.J. Chen, B. Hsieh, L. Shao, GPU-based parallel reservoir simulation for large-scale simulation problems, in: SPE Europec/EAGE Annual Conference, 2012.
[67] H. Sudan, H. Klie, R. Li, Y. Saad, High performance manycore solvers for reservoir simulation, in: 12th European Conference on the Mathematics of Oil Recovery, 2010.
[68] H. Klie, H. Sudan, R. Li, Exploiting capabilities of many core platforms in reservoir simulation, in: Paper SPE 141265 presented at the 2011 SPE Reservoir Symposium, Woodlands, Texas, 21–23 February, ed, 2011.
[69] M. Bayat, J. Killough, An experimental study of GPU acceleration for reservoir simulation, in: SPE Reservoir Simulation Symposium, 2013.
[70] Nvidia. Kepler GK110 Whitepaper, 2012. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.
[71] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, et al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Ed: Springer, 2004, pp. 97–104.
[72] H. Wang, S. Potluri, M. Luo, A.K. Singh, S. Sur, D.K. Panda, MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters, Comput. Sci.-Res. Dev. 26 (2011) 257–266.
[73] J. Kraus, An introduction to CUDA-aware MPI, ed, 2014.
[74] I.T. Center, HPC. Available: http://hpc.kfupm.edu.sa/NewHPC/Resources.html.
[75] G.V. Paolini, G.R. Di Brozolo, Data structures to vectorize CG algorithms for general sparsity patterns, BIT 29 (1989) 703–718.
[76] R. Bridson, W.-P. Tang, Ordering, anisotropy, and factored sparse approximate inverses, SIAM J. Sci. Comput. 21 (1999) 867–882.
[77] M. Benzi, M. Tuma, A comparative study of sparse approximate inverse preconditioners, Appl. Numer. Math. 30 (1999) 305–340.
[78] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2012.
[79] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Gulf Professional Publishing, 1999.
[80] C. Nvidia, Cuda c best practices guide, ed, 2014.
[81] N.I.-L.A. Gray, A. Sjöström, Best practice mini-guide accelerated clusters. Using General Purpose GPUs, ed, 2014.
[82] B. Jacob, G. Guennebaud, Eigen C++ template library for linear algebra: Matrices, vectors, numerical solvers, and related algorithms (3.2.2 ed.), 2012. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page.
[83] A.A. Awotunde, R.N. Horne, An improved adjoint-sensitivity computations for multiphase flow using wavelets, SPE J. 17 (2012) 402–417.

Please cite this article in press as: A. Zaza, et al., A CUDA based parallel multi-phase oil reservoir simulator, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.04.010