Blucher Mechanical Engineering Proceedings
May 2014, vol. 1 , num. 1
www.proceedings.blucher.com.br/evento/10wccm

10th World Congress on Computational Mechanics
8-13 July 2012 • São Paulo • Brazil

HIGH-PERFORMANCE COMPUTING IN OIL RECOVERY SIMULATION BASED ON CUDA

I. K. Beisembetov, T. T. Bekibaev, B. K. Assilbekov, U. K. Zhapbasbayev, B. K. Kenzhaliev

Scientific research laboratory of mathematical modeling of technological processes in oil and gas industry, Kazakh-British Technical University (timur_bekibaev@mail.ru)

Abstract. *In this paper the oil recovery problem and its parallel realization using CUDA architecture are considered. Results of calculation accelerations on video cards GeForce GTX 560 and GeForce 240 are presented. The factors affecting speed-up on GPU are found out. The maximum achieved speed-up is 61 times on GeForce GTX 560.*

Keywords: *Oil recovery, mathematical modeling, high-performance computing, GPU, CUDA.*

1. INTRODUCTION

The efficiency of hydrocarbons recovery is one of the major objectives of Petroleum companies during the fields development. Great number of studies, tests and computer calculations required before the wells will be drilled and the development scheme will be chosen. Computer model of hydrocarbon field is created and analyzed by modern software before and during the field exploitation. This work made routinely by the most of developers. Rule of thumb: oil can be recovered only once, but simulation can be made infinitely. Thus, software that calculate different scenarios of field development (new wells, capital investments, etc.) play important role in Petroleum Industry. One of the hydrodynamic modeling extensions is rapid calculations of recovery forecasting and history matching. Increasing number of cells in the simulation grid significantly slow down the calculations. Fast simulation on commercial software (Eclipse (Schlumberger), Tempest MORE (Roxar), VIP (Landmark), tNavigator (RFDynamics), etc.) is based on parallel computing on CPU cores of HPC (High Performance Computers) and PC using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing). This paper reviews the parallel computing of hydrodynamics on GPU (Graphical Processing Unit) of PC. It is the one of main differences between our 3D hydrodynamic simulator and existing analogues.

A great while, GPU was used with CPU only for their designated purpose - accelerate the calculations of graphics. But with increasing computation power of GPU, developers tend to apply it for CPU calculations. Nowadays, libraries that allow running the applications on the graphical processors have been developed, and the concept of GPU usage in calculations

was widely spreaded in various areas of science, from astrophysics to financial risks calculations [1-5].

Modern GPU consists of hundreds of processing cores and in some instances overcome the calculation power of CPU hundred times. These works research the calculation capabilities of GPU in oil recovery problems.

2. MATHEMATICAL MODEL

This paper reviews the hydrodynamic modeling of oil reservoir with pressure above bubble point pressure eliminating the gas from liberation. Temperature influence wasn't noticed in calculations. Thus, fluid flow in porous media could be expressed in following equations [6-8]:

$$\begin{aligned} \nabla \cdot \left[\lambda_o K \left(\nabla p_o - \gamma_o \nabla z\right)\right] &= \frac{\partial}{\partial t}\left[\frac{\phi(1-S_w)}{B_o}\right] + q_o, \\ \nabla \cdot \left[\lambda_w K \left(\nabla p_o - \nabla p_{ow} - \gamma_w \nabla z\right)\right] &= \frac{\partial}{\partial t}\left[\frac{\phi S_w}{B_w}\right] + q_w. \end{aligned}, \tag{1}$$

Equations (1) are solved using no flux boundary conditions.

3. NUMERICAL TECHNIQUES

Equations (1) were discretized using IMPES method [6-8]. Values of pressure are calculated by SOR method. Water saturation field is defined explicitly after pressure is calculated. Finally, capillary pressure and transmissibility between neighbor blocks are computed and applied for next time step.

4. PROGRAM REALIZATION ON GPU WITH CUDA

In order to use the calculation capabilities of GPU effectively, it is necessary to find the most time-consuming parts of code and adopt them by CUDA technology. Most of time in our program is spent in jobs associated with 3D arrays that have significant size. That work with cells of various 3D arrays was paralleled on GPU. On the figure below (see fig. 1), flowchart of CUDA-based program realization is shown.

CheckSchTime() – procedure, that inspects work of wells (user-defined schedules and modes). If condition of well work switching occurs, GPU data relating to wells will be updated.

PrePoissonKernel() – procedure that runs on the GPU (CUDA kernel). It computes coefficients required to solve the equation of pressure, also the kernel calculates necessary data for time step.

PoissonIterationKernel() – kernel, that executes the next iteration of the Poisson, and

determines the residual for each thread block. The procedure determines the value of the pressure in each cell of the model.

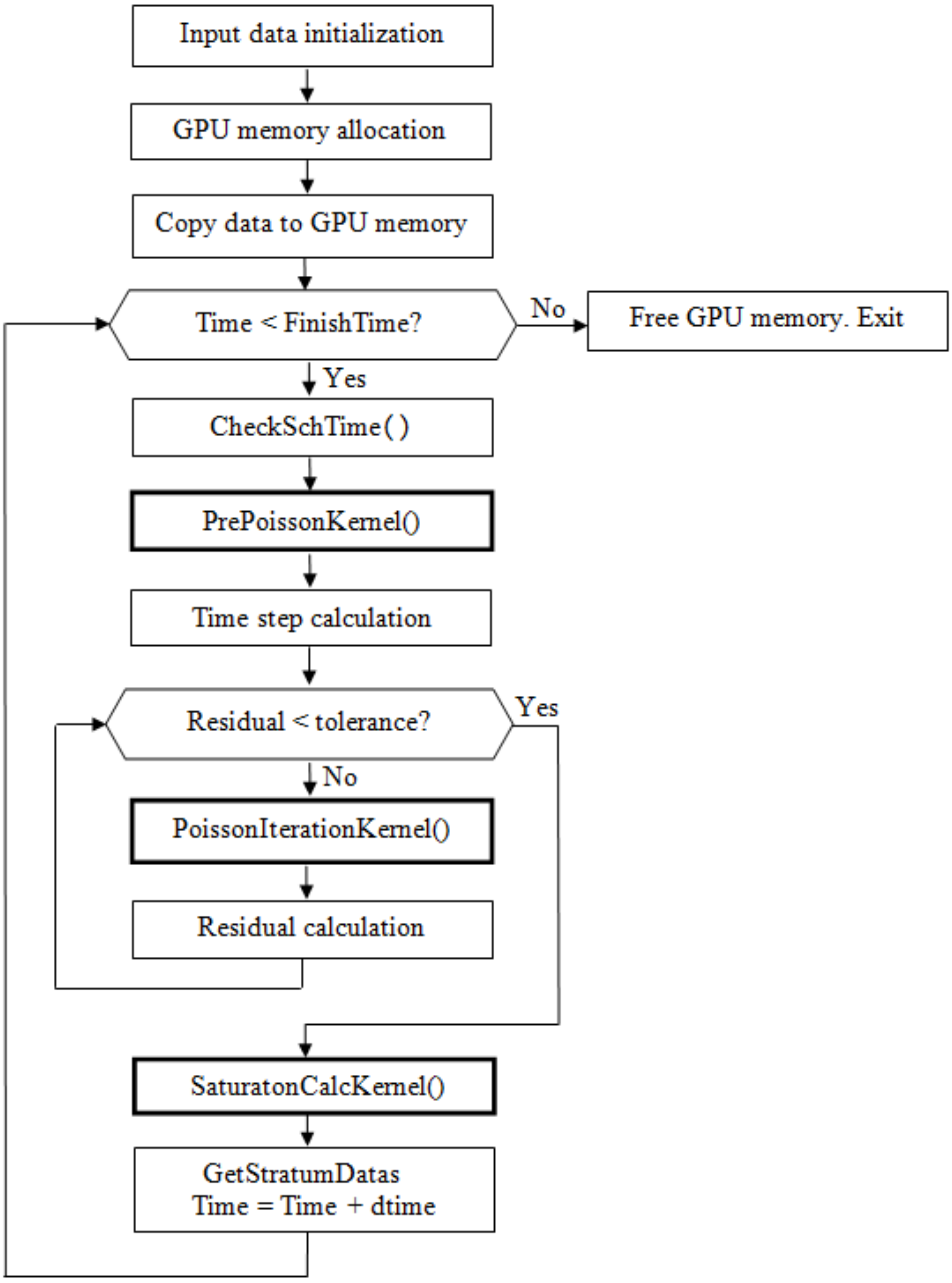


Figure 1. Flowchart representing a program realization using CUDA.

SaturatonCalcKernel() – kernel, that computes the values of water saturation in each cell, using the new values of pressure. Also the necessary parameters for well (flow rate, bottom hole pressure, water cut, etc.) are defined.

GetStratumDatas() – procedure, that's extract the field distribution of the saturation, pressure, temperature from the DRAM of video card. It takes considerable time. The procedure is only performed at user-defined time steps.

Besides, in *PrePoissonKernel()* and *PoissonIterationKernel()* parallel reduction is implemented in order to compute the values needed for current time step and residual for each thread block. The final reduction relative to the blocks is calculated sequentially on the CPU.

One of key differences between GPU and CPU is memory organization and its application. It is known, CUDA architecture has complex memory structure. In order to achieve the highest acceleration, it is necessary to accurately distribute data on various memory types. A part of data, that occupy a large amount of memory, is stored as arrays in global memory of the GPU. Each cell in the array (I,J,K) stores the value for the corresponding cell of the hydrodynamic model of the reservoir. This data are listed below:

- Main data – pressure, water saturation. They changes at each time step.
- The coefficients for the Poisson's equation. Computed in the procedure *PrePoissonKernel()* and used to determine the pressure at each iteration *PoissonIterationKernel()*.
- Invariable data - depth, the transmissibility of cells.

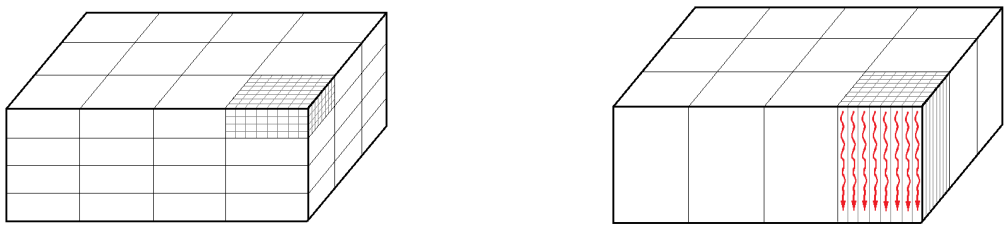
PVT and saturation function tables occupies a small volume and stored in constant memory for quick access. These tables are used repeatedly to determine the pressure and saturation for each computational cell of the reservoir.

Well data includes the values of flow rates of fluids and liquids, water cuts, bottomhole pressures for each well. The data is stored in the page-locked host memory to read it in CPU procedure and to write in GPU kernel.

Data of well management includes the type and value of work mode and information about the well perforation. It is stored in constant memory. When the well mode changes, the values of data are updated in GPU memory from CPU procedure.

Shared memory is used to store intermediate variables in the kernel. Also, thread blocks load a part of data from global to shared memory for the faster use of this data in the current kernel.

One of important aspects of algorithm creation was the choice of division of settlement area between threads. For this purpose, it was necessary to define dimension and the sizes of the block of thread. There was a choice between three-dimensional and two-dimensional division grids (see fig. 2). At three-dimensional division one thread works with one cell of model, at two-dimensional – a column of cells along an axis Z. Indexes of cells (I,J,K) are calculated concerning index of the block and index of thread.



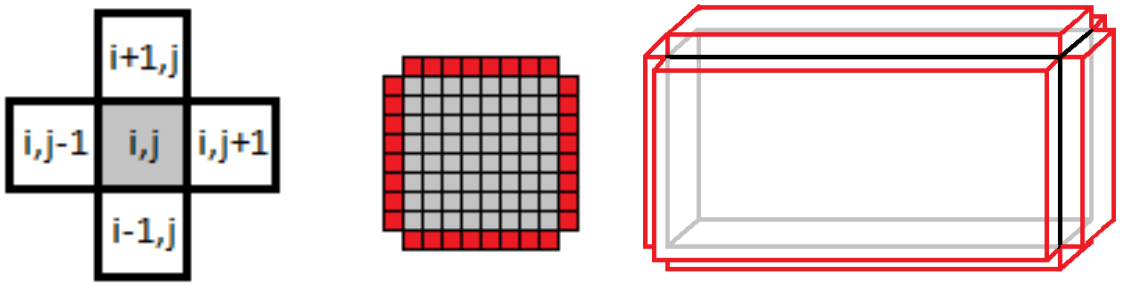
a) - 3D case b) - 2D case

Figure 2. Splitting of computational cells into thread blocks.

At calculation of each value of cell, values of neighbor cells (see fig. 3a) are required. For example, in iterative formula of the Poisson's equation data access looks like:

$$p_{ijk}^{l+1} = Ap_{i+1jk}^{l} + Bp_{i-1jk}^{l+1} + Cp_{ij+1k}^{l} + Dp_{ij-1k}^{l+1} + Ep_{ijk+1}^{l} + Fp_{ijk-1}^{l+1} + RHS. \qquad (2)$$

In order to calculate the two-dimensional region of size NX×NY it is necessary to download NX*NY + 2*NX + 2*NY values (fig. 3b), for three-dimensional region of size NX×NY×NZ - NX*NY*NZ + 2*NX*NY + 2*NX*NZ + 2*NY*NZ values (fig. 3c), because of the data access feature.



a) b) c)

Figure 3. Additional cells (red) needed to load for thread block.

Obvious at first sight, the three-dimensional configuration (see fig. 2a) is less effective. This configuration does not allow to contain a large number of cells in block (the restriction on the number of threads), which leads to the small size of the 3D thread block and, consequently, a greater ratio of "side" loads from global memory. For example, if the block has size of 8×8×8, "side" loading will be $\frac{6*8*8}{8*8*8} = 66.7\%$. In contrast, in the two-dimensional block 16x16 (fig. 3b) such loadings occupy $\frac{4*16}{16*16} = 25\%$, and in case 32×16 - $\frac{2*32+2*16}{32*16} = 18.7\%$. Extra "side" loadings – time-consuming access to global memory and branching in a code, when some threads load "side" parts and others stand idle. Besides that, such two-dimensional splitting gives the chance to use effectively the shared memory (this are described below). Therefore application of the two-dimensional splitting is more preferable.

In the consecutive program the three-dimensional loop on I, J, K is needed for processing cells of the 3D model. At two-dimensional splitting of the cells into thread blocks, each thread processes a column of cells with indexes (I,J,1) to (I,J,NZ). Therefore calculations in kernel occurs only by K-iterations. I, J indexes are defined using numbers of current block and thread. Each K-iteration computes K-layer, each thread computes a particular cell. For this purpose at first, thread loads necessary data from the global memory, then calculates necessary values, then write down it back to the global memory. Synchronization of thread in the block is required surely after data loading. The reason in, that data of the neighbor thread are necessary for calculation of new values, and the threads, belonging to a various warps and working in parallel, can will overtake each other that will lead to value calculation of a cell without a full load of all needed data. A flowchart of kernel work is presented below (fig. 4).

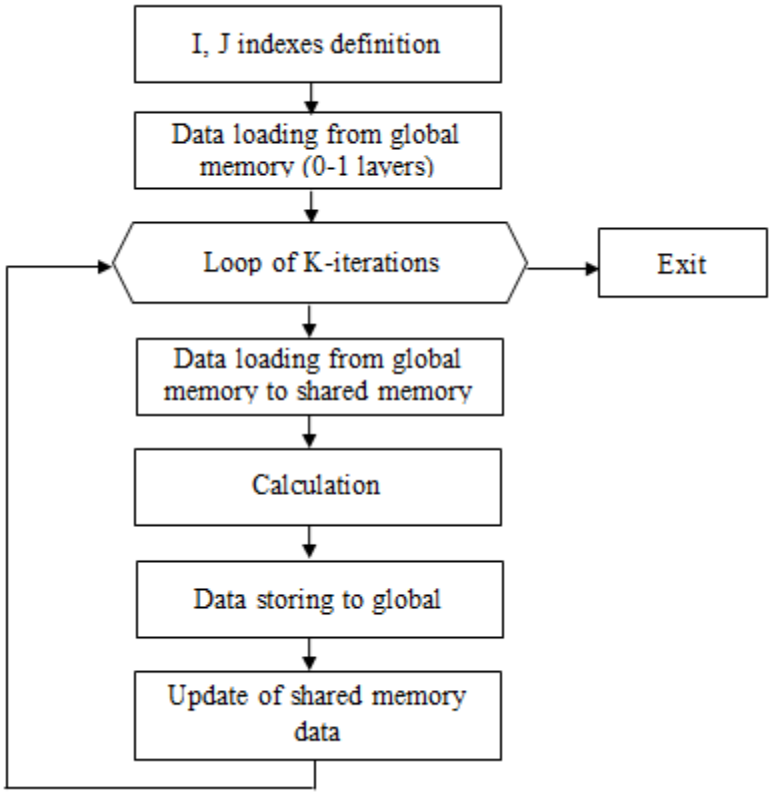


Figure 4. Flowchart of kernel work.

Data upload from the global memory to shared memory (see fig. 5) performs in two stages. At first threads load data of cells which values they will compute (see fig. 5, green cells). During this loading 16 requests to memory coalesces to one 64-byte transaction (access coalescing), because all 16 words lie in the same 64-byte segment, and i-th thread in the half-warp accesses the i-th word, located in memory one after one [9,10]. It provides a gain in time with global memory transactions. Next stage is roll-in of the “side” date (fig. 5, red and yellow cells) and some warps stand idle. Moreover, the part of data (fig. 5, red cells) can't be loaded with the access coalescing, because loaded words are in different places of memory. There are 16 separate 32 byte-transactions; it takes up twice more time than upload of a green part of the array. «Side» loading (especially red part) is one of slow kernel parts on speed.

In order to calculate values on a layer K (by Z-axis), values on layers K-1 and K+1 are needed, (additional layers above and below model are entered for this purpose). When next layer starts to be computed in GPU, layer K+1 becomes K, and previous layer K becomes K-1, the values are stored on shared memory. Using shared memory allows to decrease reading from global memory, because to calculate new layer, there is no need to load three layers, it is enough to load only K+1, and the others already stored from last iterations by K. In end of the each iteration update of shared memory array occurs (previous data of K-1 changed by data of K, previous data of K changed by K+1).

However, working with the shared memory array, there can be a situation when two and more threads address to one bank of memory (bank conflict) [9,10]. It decreases performance of access to this type of GPU memory. In order to avoid the bank conflict, one ficti-

tious column is added. Owing to it, the banks in the two-dimensional array are distributed so that all threads of a half-warp at the same time address to different banks.

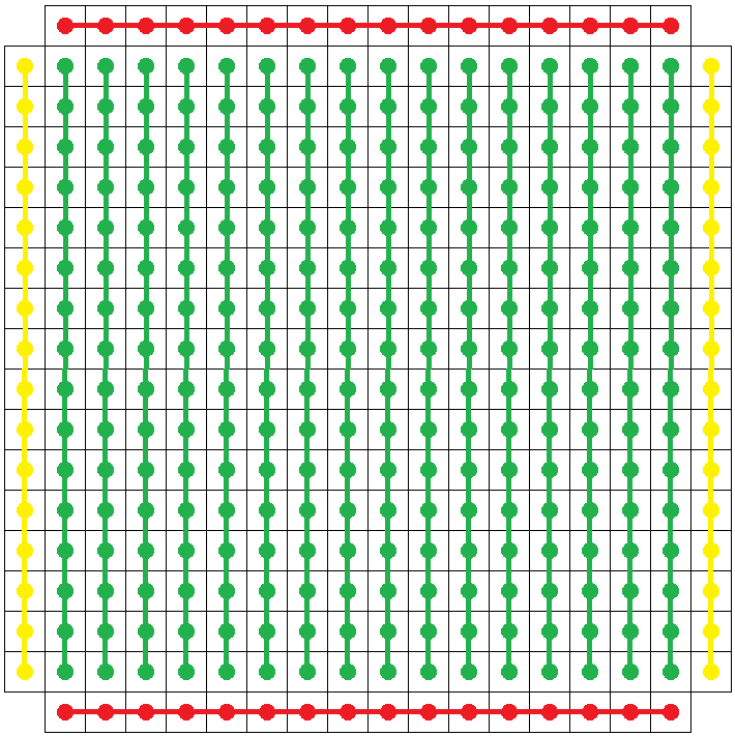


Figure 5. Data loading by half-warps to 2D array in shared memory.

The figure 6 shows an example of 2D array located in shared memory when thread block size is 16×16. Due to «side» data, minimum size of array must be 18×18. At such size (fig. 6, a) the concurrent access of half-warp (yellow color) will lead the 2-way bank conflict (for example, banks with number 1,3,5,7,9,11,13,15 will be read/written by 2 thread). At the size 18×19 (fig. 6, b), there is no bank conflict because of an additional unused column (red).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3

a) - 18×18 - 2-way bank conflicts

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4
6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10
12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13
15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3
5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6
8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12
14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2
4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5
7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8
10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11
13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1
3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4

b) - 18×19 - no bank conflicts

Figure 6. Banks distribution in 2D array of size.

Comparing thread blocks of size 16×16, 32×16 and 32×8, it should be noted that at global memory allocation it is necessary to make array padding to satisfy the coalescing condition (association of transaction to memory) when half-warps access to DRAM. 32×8 or 32×8 tiling requires more memory offset than 16×16 tiling. Because of the smaller sizes, the thread block 16×16 is better scaled under the various sizes of a grid of model along an axis X (or Y). However, the blocks 32×16 or 32×8 are more suitable for video cards with compute capability 2.x because in this architecture a warp (not a half-warp) request to global memory, i.e. access takes one transaction, in contrast, warp in 16×16 tiling needs 2 transactions.

Also, it is necessary to take into account the occupancy of streaming multiprocessor (SM). It means the number of thread blocks, that one SM can execute, which depends on number of active warps in SM. In this work it was experimentally established that on computing capability 2.x GPU the program gives the highest acceleration when 32 of 48 warps (which can be executed on one SM) are active, in other words optimal occupancy is 67% for various block sizes (see fig. 7). In present algorithm, maximum occupancy of SM is not effective. This is due to best latency hiding in process of warp access to global memory and a large number of registers on a thread for intermediate calculations by contrast to the maximum occupancy.

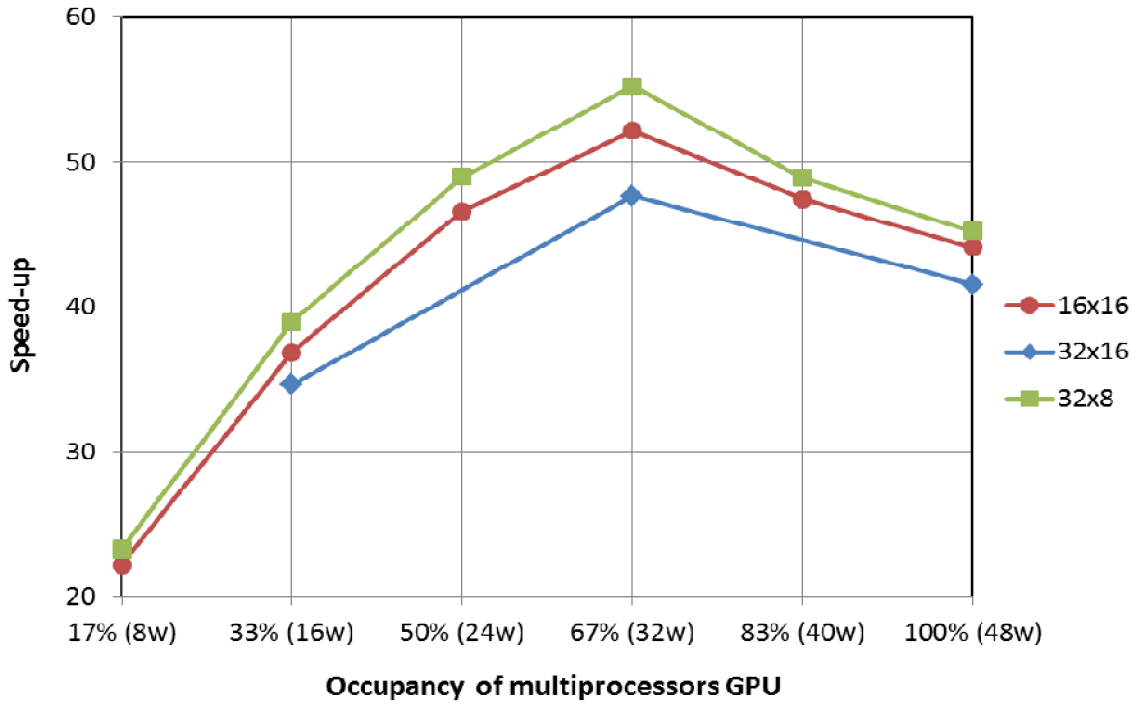


Figure 7. Acceleration at various occupancy of SM and sizes of thread block.

Most of scientific calculations usually are performed with the double precision floating-point variable (FP64). Not all GPU supports computation with the FP64. It was not necessary in the past, because single precision is enough for operations with the graphic. Computation with FP64 on CUDA is supported by new generation of NVIDIA video card with Compute Capability 1.3 and higher. However, double precision performance is essentially lower than single. GPUs of the Fermi architecture are more suitable for this:

streaming multiprocessors have more special blocks for operations with FP64 (Double Precision Units), they have improvements related to the shared memory (it has 32 memory banks, there is no 2-way bank conflict) and to the global memory (two memory requests, one for each half-warp, if the word is sized of 8 bytes i.e. FP64).

On our video card GTX 560, operation performance with FP32 is 1345 GFLOPS, with FP64 is 168 GFLOPS. Our experiments showed, on this GPU calculations with double precision in contrast of single precision concede in speed in 1.6-1.8 times. For our computation single precision is enough, because results with FP32 and FP64 differ for 1-3%. Therefore, for carrying out a series of calculations, single precision was used on the consecutive and parallel version of the program was used.

5. NUMERICAL EXPERIMENTS. RESULTS AND DISCUSSIONS

Numerical experiments are performed for two cases. In case 1, calculations took place on the field test model and was aimed to find the influence of cells distribution on acceleration (various cells number or NX, NY, NZ proportion by X, Y, Z axis at constant total number of cells, "grid size ratios"). In case 2, calculations was made on East Moldabek field model. Below, models descriptions are presented.

In the market of video cards, supporting the CUDA technology (NVIDIA products), there exist a varaity of GPU, with various computing characteristics and architecture. In this article, parralel calculations results of two GPU - GT 240 (Compute Capability 1.2, 96 cores) and GTX560 (Compute Capability 2.1, 336 cores) - are represented. Consecutive version of the program was runned on Intel Core i7 2600 CPU (3.40 GHz, DRAM Frequency 667 MHz).

Case 1. Homogenous cells with permeability of 2900 mD and 33% porosity. Initial pore pressure - 100 bars, water saturation - 20%. Heterogeneity of characteristics through reservoir space does not affect the software effectiveness. Model has 2 wells: 1 production well and 1 injection well, located on opposite corners. (see fig. 8).

On the figure below (fig. 9), relationship between NX, NY, NZ proportion at constant cells number and calculations acceleration is shown. Thread block of 16x16 was used. In all experiments, total number of cells was 921 600. Comparison made at NZ values of 1 to 3600 and NX*NY*NZ=921600.

Two factors influence the acceleration at these experiments.

The first is the usage of streaming multiprocessors, i.e. the distribution of thread blocks to all SMs. For example GTX 560 has 7 SM, each one can execute 4 thread blocks, i.e. for one certain step they can process 28 blocks. Therefore, it is important, that in splitting of model cells into thread blocks its number will be divisible by 28. For example, the model of size NX=96, NY=96 splits into 36 blocks 16×16, and at such block number on the first step all of the 7th multiprocessor will work, counting 28 blocks, on the second step the remained 8 blocks will be computed by only 2 SMs, and the others 5 SMs will stand idle. In other words, SMs usage will be $\frac{36}{2*28} = 64\%$. Obviously, the usage is more, so the acceleration on GPU is higher. If the number of blocks is much less than 28, only few SMs will be used on the first step, i.e. the conveyor from 7 SMs will not work full efficiently. And as a result, speed-up will dramatically decrease at small NX and NY values. It should be noted that divisibility of

block number on 28 is important only at small NX, NY.

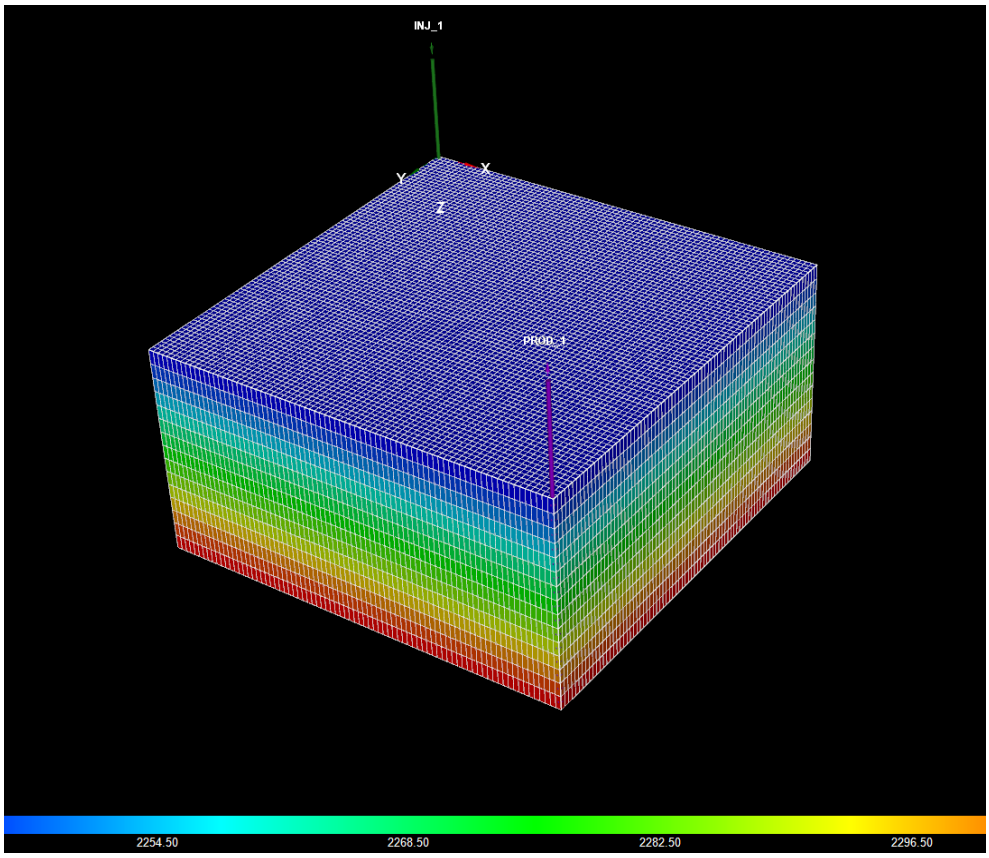


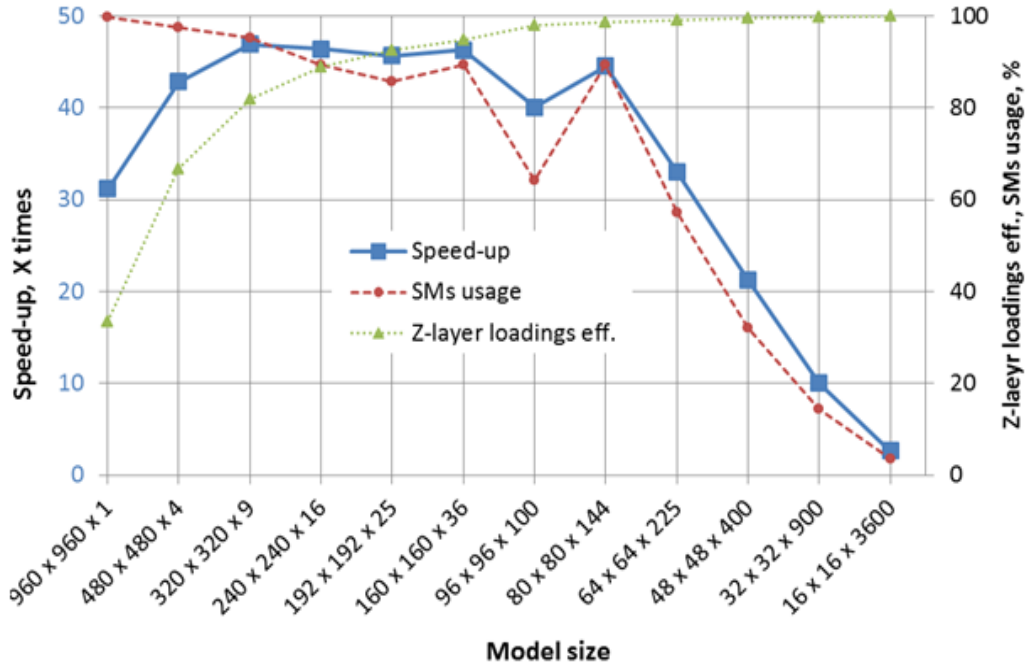
Figure 8. Test model for case 1.

At great values of NX, NY (i.e. at a large number of blocks) divisibility on 28 is not so important and does not essentially influence the acceleration because there will be a plenty of 28 blocks calculation steps. The unoccupied SMs could remain only on the last step, thus total use of all multiprocessors will tend to 100%.

The second factor affecting the speed-up is related to the NZ. Owing to use of shared memory in thread block, only NZ+2 layers are loaded from global memory to calculate NZ layers. It is known that, accesses to DRAM are carried out very slowly on GPU. Therefore the number of transaction to DRAM is critical. At small values NZ, amount of "superfluous" transactions essentially in comparison with NZ. For example, for calculation model with NZ=1, loading of 3 layers is required, for NZ=2 – 4 layers, and for NZ=100 – 102 layers. Ratios of the calculated layers number to loaded are 33%, 50% and 98% accordingly. Obviously, if this indicator is higher, the acceleration will increase. It explains the less speed-up of GPU calculation of models with small NZ.

Thus, at identical number of grid cells, model with the balanced sizes NX, NY, NZ has the highest speed-up on GPU program. Acceleration considerably decreases, if NX, NY is too small.

a)



b)

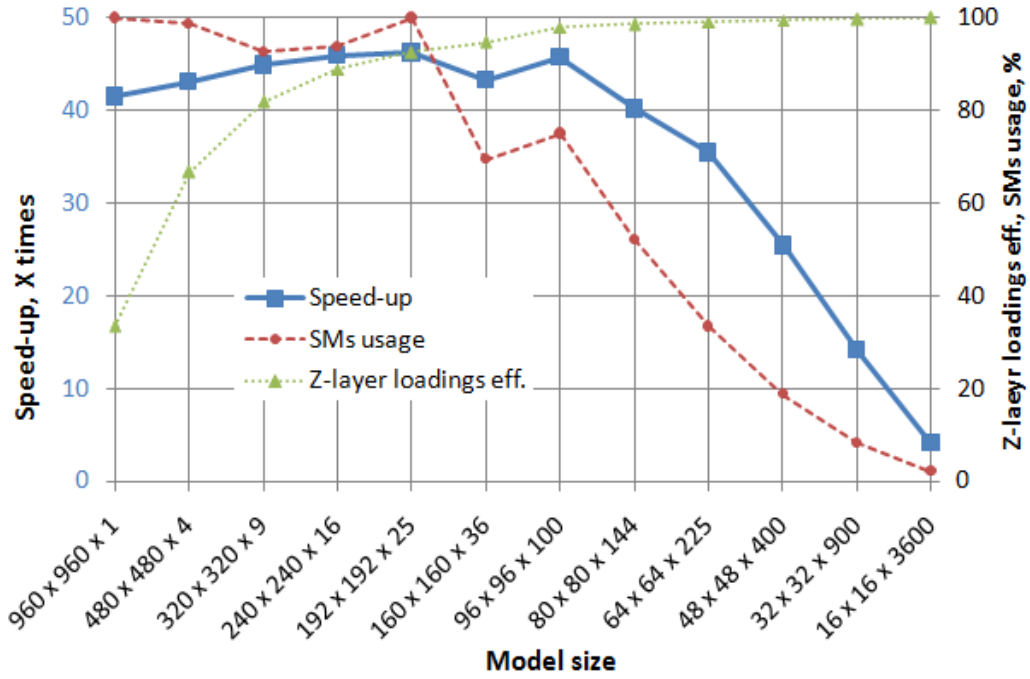


Figure 9. The dependence between the acceleration and grid size ratios:
a) - on GTX 560; b) - on GT240.

Figure 10 shows a dependence of the acceleration from the total number of grid cells in hydrodynamic model, where NX=NY=$\frac{16}{3}$NZ. In this case total number of cells changed from 96000 to 6144000 and thread block 16×16 was used.

Figure 10 a) shows, increase in total number of cells grows the acceleration. After two million cells acceleration is stabilized and equal to 52-55 times. Figure oscillation, especially in the beginning, can be explained by various coefficients of SMs usage that was discussed before. Small speed-up at low amount of cells is explained by smaller fraction of parallel part

of program comparing with the sequential part of program. (Parallel part: various values calculation of model 3D grid; sequential: well mode recalculation and limits verification, kernel launch, results output).

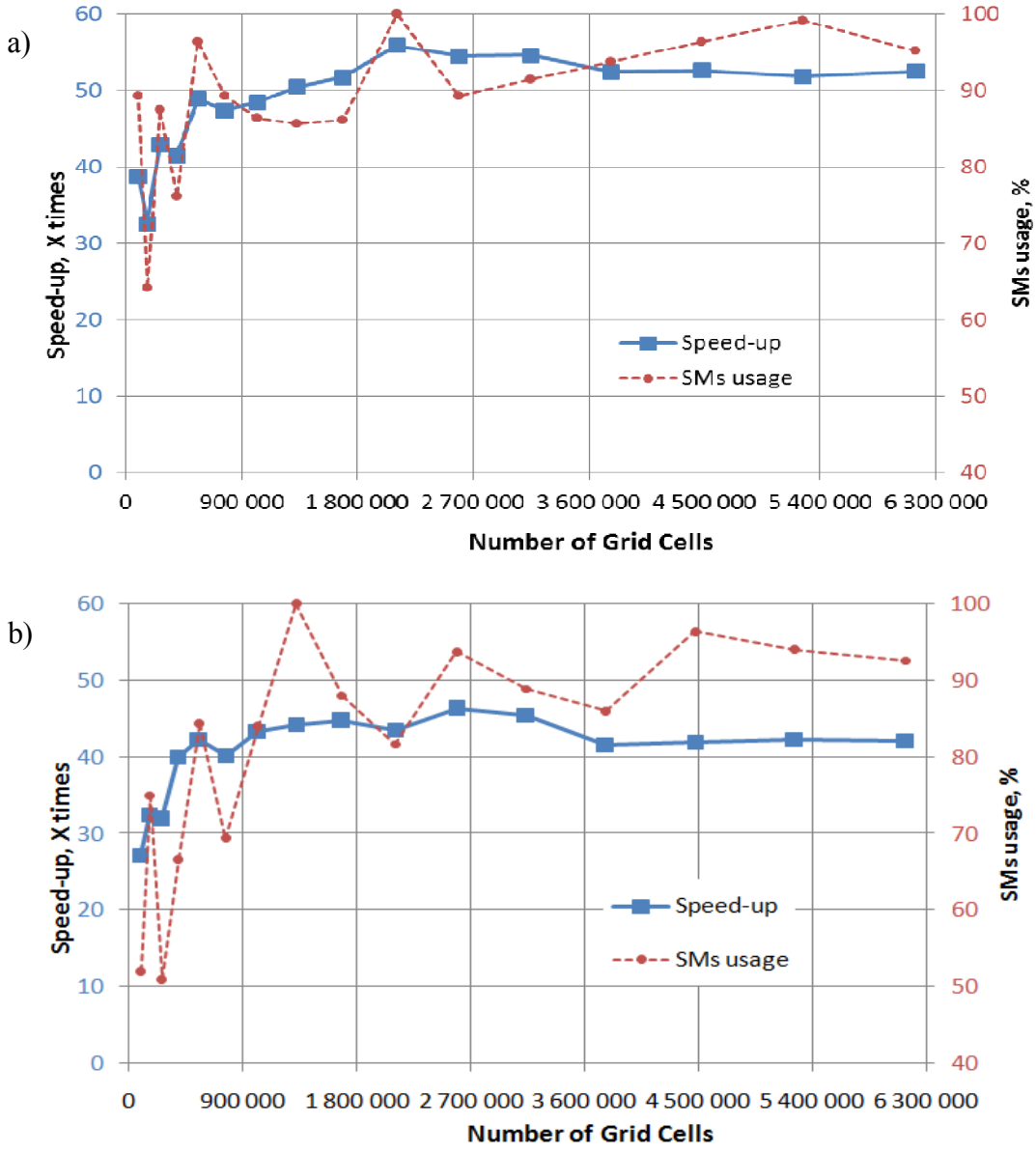


Figure 10. Dependence between the acceleration and total number of grid cells: a) - on GTX 560; b) - on GT240.

Maximum speed-up of 60.4 times was taken on grid of 2 million and more cells, using the thread block 32×8.

It is necessary to mention some factors which negatively affect acceleration at the current implementation of the program:

- Divisibility of NX, NY by sizes of thread blocks (16×16, 32×16, 32×8). If NX or NY is not divisible by block size, NX and NY will be rounded up, amount of blocks will increase and in some boundary blocks threads will be idle. For example, model with NX×NY=48×48 has 9 blocks, model with NX×NY=48×49 has 12

blocks (such as 48×64). However, for large NX and NY, it is becoming less important.

- Schedule of wells work. Frequent mode switches of wells increase data transfer between the RAM and the GPU memory.
- Existence of inactive cells. The block, which has inactive cells, is computed for the same time, as well as the block, which has all active cells. Whereas in consecutive realization, inactive cells reduce number of iteration at looping on a three-dimensional grid that reduces time of consecutive calculation.

Analyzing results of calculations on GT 240 and comparing them to results of GTX560, it is possible to notice that acceleration depends on the same factors, said about before. In spite of that GT240 has 96 cores (instead of 336), it concedes on productivity only a little, reaching the maximum acceleration in 46.27 times. The matter is that the multiprocessors of this architecture contain 8 cores (instead of 48) and consequently, GT240 has 12 SM (instead of 7). The more number of cores in SM conducts to faster speed of arithmetic operations execution in the thread block. The more number of SM conducts to more number of parallel executing blocks on all SMs.

It was experimentally found out that optimum occupancy of SM for both GPU is 4 thread blocks of size 16×16 that made 67% for GTX560 and 100% for GT240. To better hide a the latency of memory accesses GTX560 doesn't use a full occupancy of SM. That fact, that the device with Compute Capability 2.x has large volume of shared memory, is not advantage in this case. 16 Kb is enough for shared memory capacity of four thread blocks on a SM and it is available to devices with CC 1.x. However, on CC 2.x access to global memory occurs a little quicker owing to the L1 and L2 cache. The increase in registers number at a thread on CC 2.x allows avoiding use of the slow local memory. Except architectural distinctions, GTX 560 has better technical characteristics: clock rate of cores – 550 MHz (GT 240) и 810 MHz (GTX 560), maximum memory bandwidth – 57.6 Gb/s (GT 240) и 128 (GTX 560). Therefore, both the GPU has its advantages and disadvantages that affect their performance.

Therefore, GPU with Compute Capability 2.x is more suitable for parallel calculations, but also it is necessary to consider technical characteristics (such as number of SM, clock rates of cores , memory frequency and bandwidth) of the particular video card.

Case 2. In this case, there is considered East Moldabek field model (see fig. 11). The model consists of 91476 (36×77×33) grid cells. Formation volume factor of water, water viscosity and porosity relations from pressure are set up in the following equations for East Moldabek field:

$$B_w = 1.02\left[1 + 3.25 \cdot 10^{-6}(p_w - 20) + 5.28 \cdot 10^{-12}(p_w - 20)^2\right]^{-1}, \quad (3)$$

$$B_w \mu_w = 1.029\left[1 + 2.39 \cdot 10^{-6}(p_w - 20) + 2.85 \cdot 10^{-12}(p_w - 20)^2\right]^{-1}, \quad (4)$$

$$\phi = \phi_0\left[1 + 5 \cdot 10^{-5}(p_o - 39) + 1.25 \cdot 10^{-9}(p_o - 39)^2\right]. \quad (5)$$

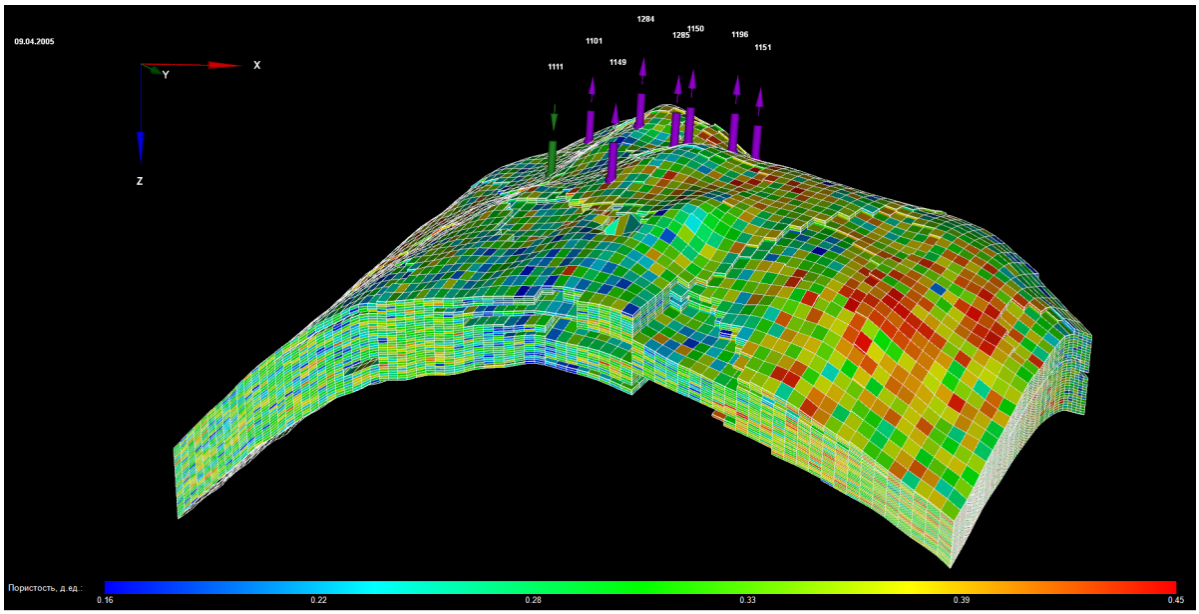


Figure 11. Porosity distribution of East Moldabek field.

East Moldabek field's oil and water density at standard conditions are 889.5 and 1000.1 kg/m^3 respectively. Horizontal permeability varies from 503.2 mD to 4240.1 mD, while vertical permeability ranges are 53 mD and 420 mD. Porosity changes in range of 17-40%. The Fig. 12 shows the initial water saturation distribution. Dependence of relative phase permeability and capillary pressure from water saturation is represented in Fig. 13. Fig. 14 shows the oil PVT properties.

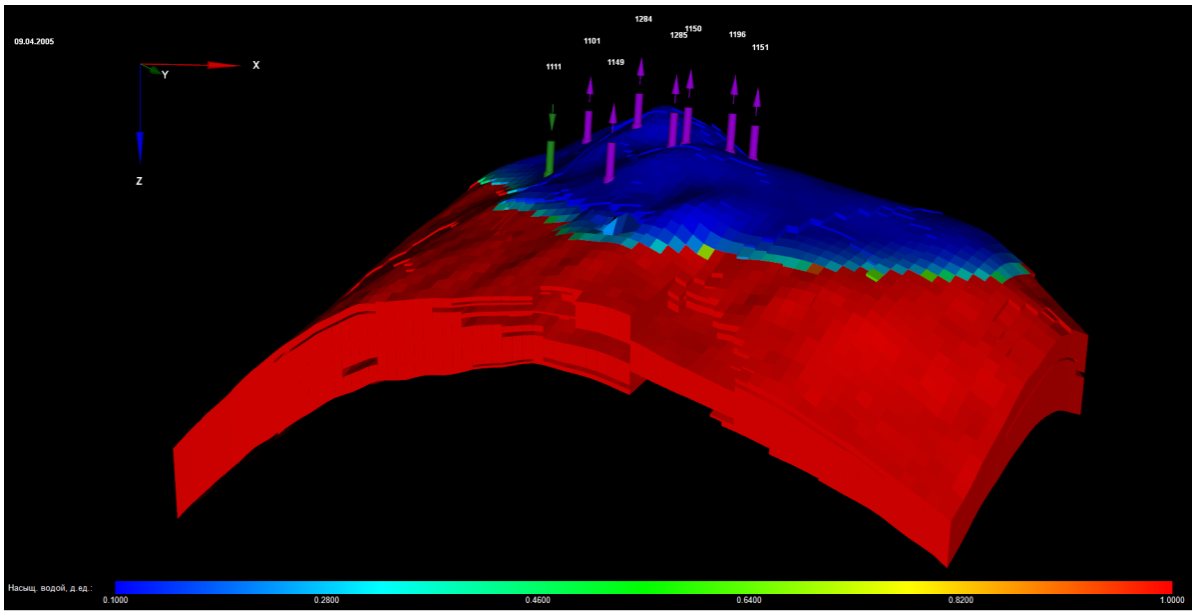


Figure 12. Initial water saturation distribution of East Moldabek field.

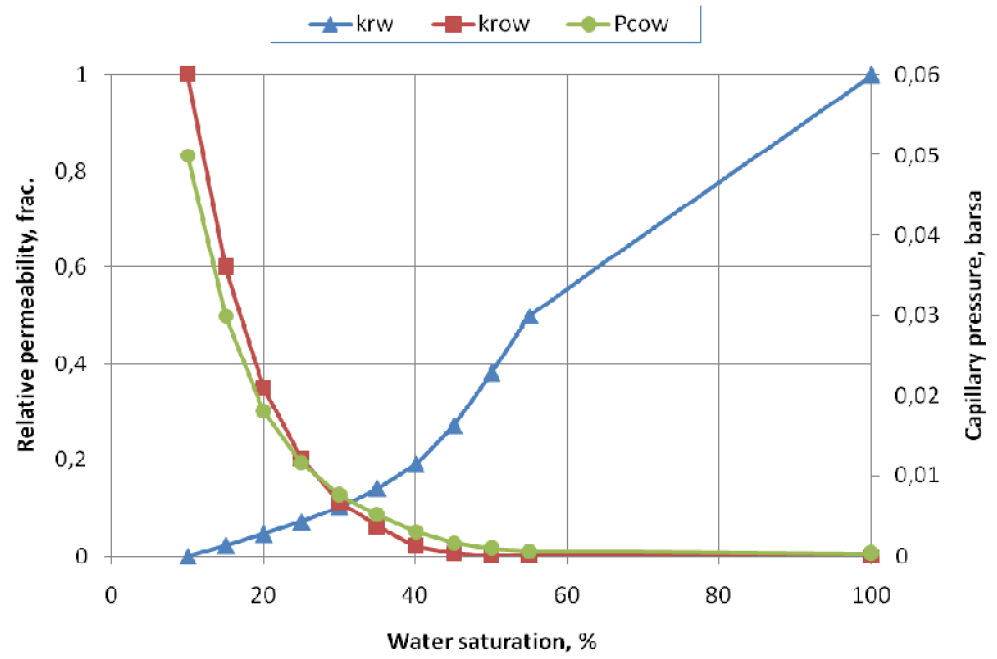


Figure 13. Relative permeabilities and capillary pressure for case 2.

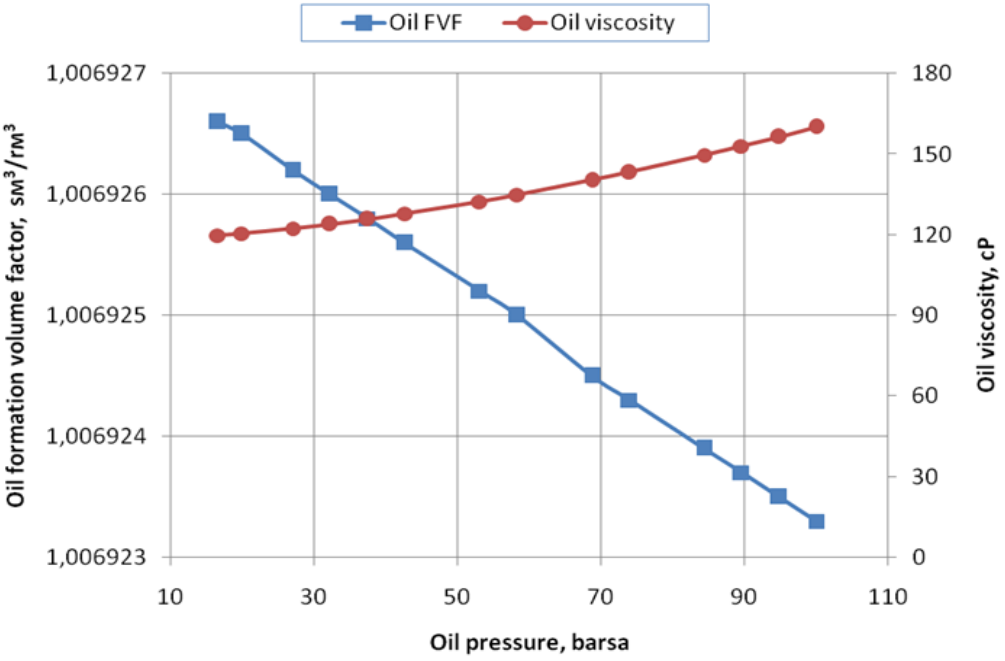


Figure 14. Oil PVT properties for case 2.

The carried out calculation consisted of 100 iterations on time that totally composes 450 days of forecast. Operating time of the parallel version of the program is 5.3 sec., and 33 times acceleration is gained on GTX 560 (in comparison of the unparallel version of the program). Calculations results of two versions were completely identical. Less calculation speed-up comparing with the maximum achieved acceleration (60.4 times), can be explained by the small size of model and as a result, the underloading of all sreaming multiproccers of GPU.

6. CONCLUSIONS

In this work transposition and optimization of the program of oil recovery simulation is carried out from the consecutive version to parallel version on GPU using CUDA technology. Obtained results were analyzed and factors affecting to performance are determined for this problem. Video cards GeForce GTX 560 and GT240 were used to run the parallel calculations. Consecutive calculations were spent on the central processor Intel Core i7-2600.

Process of transposition consisted of extraction from algorithm the several parts which can be carried out with the maximum degree of parallelism, and rewriting these parts for GPU kernels. For these purposes there were chosen parts of a code, which are related to the calculations of all cells values in a three-dimensional grid of model, because computation of this cells spend the main time in the consecutive program, especially at the big-sized model. Input data for calculations was divided into various classes of CUDA memory, depending on their volume, frequency and need of use. All model cells have been split into two-dimensional regions of thread blocks, each thread of them calculates cells with indexes from (I,J,1) to (I,J,NZ). Two-dimensional model splitting showed more efficiency comparing with three-dimensional splitting. It was found that the optimum size of thread blocks for acceleration – 32×8, however the 16×16 tiling is better scaled under the various sizes of model grid and slightly surrenders in performance. Therefore for further calculations the block 16×16 was used.

A series of calculations on the test model (with different ratios NX, NY, NZ relative to each other and the total number), and the model of the real field East Moldabek were executed. Analyzing the results of the calculations and identifying the main factors affecting the performance, it is possible to predict the acceleration value of the model in its size (figures 7, 9-10). Calculations show that the highest speed-up on GPU is reached on models with large grid size (2 million cells and more). Frequent mode switches of wells and inactive cells negatively influence acceleration. Incomplete loading of GPU multiprocessors essentially reduces performance that occurs during the calculations of models with small NX, NY values.

Comparing performances of GPUs with two different architecture, it is found out that, GPU with Compute Capability 2.x is more suitable for parallel calculations, but also it is necessary to consider technical characteristics (such as number of SM, clock rates of cores , GPU memory frequency and bandwidth) of the particular videocard. The maximum achieved speed-up is 60.4 times on GeForce GTX 560. It is clear that more productive GPU (Tesla series based on the Fermi Architecture having more processing cores and more memory bandwidth), acceleration of calculation will be much higher.

In conclusion, it was found out, that algorithm of the IMPES method for oil recovery simulation is perfectly suitable for architecture of CUDA. Considerable acceleration (from 25 to 60.4 times) was achieved. Acceleration of water flooding calculations on GPU can be competitive alternative to use of expensive high-performance clusters based of CPU.

7. REFERENCES

[1] Buatois L., Caumon G., Levy B., "Concurrent number cruncher: a GPU implementation of a general sparse linear solver". *Int. J. Parallel, Emergent and Distributed Systems.*24, 205-223, 2009.

[2] http://www.seismiccity.com/Technologies.html.

[3] http://www.hanweckassoc.com/.

[4] http://www.ks.uiuc.edu/Research/namd/.

[5] Zverev Ye., Novozhilov Yu., Mikhalyuk D. "Acceleration of engineering calculations with GPU NVIDIA Tesla in ANSYS Mechanical". *Engineering Tech. J.* 33-38, 2011.

[6] Aziz K., Settari A. "Petroleum Reservoir Simulation". *New York: Elsevier*, 406 p., 1979.

[7] Chen Z., Huan G., Ma Yu. "Computational Methods for Multiphase Flows in Porous Media". *Philadelphia: SIAM*, 549 p., 2006.

[8] Fanchi J. R. "Principles of Applied Reservoir Simulation". 2nd Edition. *Houston: Gulf professional publishing*, 357 p., 2001.

[9] "NVIDIA CUDA C Programming Guide". Version 4.0. *Santa Clara CA*: *NVIDIA*, 187 p., 2011.

[10] Kirk D., Hwu W. "Programming Massively Parallel Processors". *Burlington: Elsevier*, 75 p., 2010.