Optimization Procedures During Parallelization of Specialized Software for Fluid Flow Simulations

Filipović, Nenad; and Đukić, Tijana

Abstract — ***Modern fluid flow simulations can be extremely complex and computationally demanding. Using GPU devices (Graphics Processing Unit) they can execute up to several tens of times faster and simulations can be observed interactively. In this study the basic principles of GPU programming are applied to the implementation of lattice Boltzmann (LB) method. The software that was developed based on the basic LB equations is parallelized and a discussion is given about certain improvements made on the initial implementation.The developed software was tested on a Tesla GPU device and significant speed-up is obtained, when comparing to the traditional version of the software. Fluid flow simulations in the field of biomedicine that needed up to a few hours to be performed, can now be finish in just a few minutes.***

Index Terms — ***graphics processing unit, lattice Boltzmann method, CUDA architecture, execution time comparison, parallelization speed-up***

1. INTRODUCTION

Scientific methods used to include only observation, hypothesis and experimentation, but in the past 20 years computer simulations have become a new scientific paradigm [1]. Modern methods of computer scientific simulation of various phenomena, such as fluid flow simulations, can be extremely complex. Computational demands of these applications are high and large computational resources are required in order to perform these simulations. GPU devices (Graphics Processing Unit) represent a good choice for applications with high requirements in terms of computational resources. These devices have drawn much attention since they could be used in a wide range of general-purpose applications [2,3]. GPU devices execute hundreds or even thousands of threads simultaneously and thus accelerate calculations. The greatest benefit is that GPU device works as a coprocessor of the main computer, and together they form an effective system. Also it is not necessary to introduce large modifications in existing computer programs, thanks to a specially designed CUDA architecture (Compute Unified Device Architecture). With the help of GPU devices, programs that used to execute for a few days or hours, can now execute up to several tens of times faster. This way, simulations of many different phenomena can be observed interactively. There are many examples of successful application of GPU devices in different areas [4-6].

The problem of fluid flow simulation can be numerically solved using a wide range of methods, such as a continuum based finite element method [7] or discrete methods like dissipative particle dynamics [8]. This paper considers fluid flow simulations using a discrete method called lattice Boltzmann (LB) method. This method observes the fluid as a set of fictitious particles and by studying the dynamics of these particles (the collisions between them) the fluid flow is modeled on the macroscopic level. The greatest advantages of LB method are the simplicity of implementation and the possibility of parallelization.

Nowadays, in order to obtain high-accuracy in simulations of fluid flow in complex geometries, it is necessary to define a fine mesh, which results into many thousands of nodes. And if it is expected to get results in a reasonable computing time, it is necessary to parallelize the software. Different CFD solvers were parallelized using GPU devices and the obtained speed-ups were published in literature, including more traditional methods [9,10], but also lattice Boltzmann method [11-13].

In this study the basic principles of GPU programming are applied to fluid flow simulations, namely to the implementation of lattice Boltzmann method. The first implementation is further optimized and the improvements in terms of obtained speed-up are discussed. The developed software is tested on a specialized Tesla GPU

Manuscript received April 20, 2013. We acknowledge that the results of these simulations have been achieved using the PRACE Research Infrastructure resource CURIE based in France at TGCC (Très Grand Centre de calcul du CEA).

N. Filipović is with the Faculty of Engineering, University of Kragujevac, Serbia. Also, he is with Harvard University, Boston, USA and with BioIRC R&D Center, Kragujevac, Serbia (corresponding author, phone: +38134334379; fax: +38134334379; e-mail: fica@kg.ac.rs).

T. Đukić is with the Faculty of Engineering, University of Kragujevac, Serbia. Also, she is with BioIRC R&D Center, Kragujevac, Serbia (e-mail: tijana@kg.ac.rs).

device and a speed-up of 21 times is obtained, when comparing to the traditional version of the software.

The paper is organized as follows. In Section 2 the theoretical background of lattice Boltzmann method and the final form of equations used in numerical implementation of LB method are presented. Some details of the implementation are the subject of Section3. Parallelization of the software is explained in Section 4. Section 5 concludes the paper.

2. THEORETICAL BACKGROUND OF LATTICE BOLTZMANN METHOD

The lattice Boltzmann method belongs to the class of problems named Cellular Automata (CA) and observes the physical system in an idealized way, so that space and time are discretized, and the whole domain is made up of a large number of identical cells [14]. Single distribution function is defined that represents the probability for particles to be located within a certain space element. This function in a specific lattice cell depends on the state of neighboring cells and it has an identical form for all cells. The state of all cells is updated synchronously, through a series of iterations, in discrete time steps. The derivation procedure of the lattice Boltzmann method can be found in literature [15,16] and only the initial and final equations will be given in this paper.

In the presence of an external force field , one can write a distribution function balance equation – Boltzmann equation:

$$\frac{\partial f}{\partial t}+\mathbf{v}\frac{\partial f}{\partial \mathbf{x}}+\frac{\mathbf{g}}{m}\cdot\frac{\partial f}{\partial \mathbf{v}}=\Omega \tag{1}$$

where Ω is the collision operator.

Since this operator is represented using a very complex expression, a simplified model is introduced, initially proposed by Bhatnagar, Gross and Krook [17]. This model is known as the single relaxation time approximation or the Bhatnagar-Gross-Krook (BGK) model. Operator Ω is defined as follows:

$$\Omega=-\frac{1}{\tau}\left(f-f^{(0)}\right) \tag{2}$$

where τ is the relaxation time (the average time period between two collisions) and is the equilibrium distribution function, the so-called Maxwell-Boltzmann distribution function.

Finally, BGK model of the continuous Boltzmann equation is given by:

$$\frac{\partial f}{\partial t}+\mathbf{v}\frac{\partial f}{\partial \mathbf{x}}+\mathbf{g}\frac{\partial f}{\partial \mathbf{v}}=-\frac{1}{\tau}\left(f-f^{(0)}\right) \tag{3}$$

The original BGK Boltzmann equation is continuous in space domain and is related to continuous velocity field. This form is not suitable for numerical implementation. In order to develop a program that numerically solves this equation on a computer, the equations have to be previously discretized. After discretization, the macroscopic quantities are evaluated in terms of the distribution function, as weighted sums over a finite number of discrete velocities.

The discretized equation that is numerically solved in software based on LB method is given by:

$$f_i(\mathbf{x}+\mathbf{v}_i,t+1)-f_i(\mathbf{x},t)=-\frac{1}{\bar{\tau}}\left(f_i(\mathbf{x},t)-f_i^{(0)}(\rho,\mathbf{u})\right)+\left(1-\frac{1}{2\bar{\tau}}\right)F_i \tag{4}$$

where $\bar{\tau}$ is the modified relaxation time, introduced to provide more appropriate time discretization and better numerical stability of the solution.

When LB method is implemented on a computer, this equation is most commonly solved in two steps – the collision step and the propagation step. Two values of the distribution function can be defined - f_i^{in} and f_i^{out}, that represent the values of the discretized distribution function before and after the collision, respectively. The mentioned steps are expressed as follows:

Collision step:

$$f_i^{out}(\mathbf{x},t)=f_i^{in}(\mathbf{x},t)-\frac{1}{\bar{\tau}}\left(f_i^{in}(\mathbf{x},t)-f_i^{(0)}(\rho,\mathbf{u})\right)+\left(1-\frac{1}{2\bar{\tau}}\right)F_i \tag{5}$$

Propagation step:

$$f_i^{in}(\mathbf{x}+\mathbf{v}_i,t+1)=f_i^{out}(\mathbf{x},t) \tag{6}$$

Each one of these steps must be applied to the whole system (to all nodes of the lattice mesh, i.e. to all particles) before the next step starts. The step that considers collisions is a completely local operation (the equations for every node are independent and not coupled), while the step that considers propagation takes into account only the currently considered node and a few closest neighboring nodes. This is why the described scheme and LB method are particularly convenient for parallelization.

3. IMPLEMENTATION OF LB METHOD

A specialized software based on lattice Boltzmann method is developed. The basic idea of the implementation of LB method is to divide the observed domain on a certain number of identical elements that are also called cells. For two-dimensional problems the type of mesh that is used is denoted by D2Q9, because it is a two-dimensional domain. For every node of this mesh 9 components of distribution function are defined. Figure 1 shows the domain in two-dimensional space and the lattice mesh, and also shows the directions of the discretized distribution function for one node of the mesh.

The program for the simulation of fluid flow, based on LB method is written in programming language C++. There are functions that were specifically developed to manipulate data related to nodes individually, to the entire mesh (within the mesh the interactions between nodes are considered), as well as the entire simulation (that controls all other activities). All of these functions are implemented within several classes, such that there is particular class that defines a node, the mesh and the simulation. It is important to

emphasize that there must exist a loop of iterations within the program, since the problem is solved in a large number of time steps, until the steady state is reached.

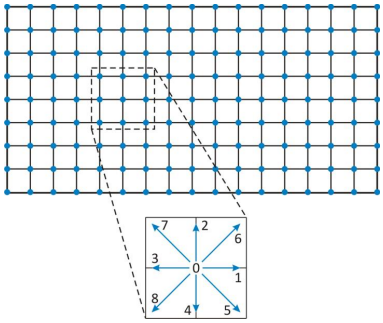

Figure 1 - Two-dimensional mesh and discretized distribution function [18]

The developed software was used to simulate diverse problems in the field of biomedicine. The software was initially tested on simple examples like straight stationary flow between two parallel walls. But the software is primarily used to simulate more complex problems of fluid flow, e.g. simulation of fluid flow through an artery with bifurcation and fluid flow through a human aorta.

4. PARALLELIZATION OF THE LB SOLVER

GPU devices are especially adapted to solve problems that can be described as processing a large number of arithmetical operations over a large data set. In other words, when the same series of operations needs to be performed on a large data set in parallel, with a small number of memory operations and a large number of arithmetical calculations – it is suitable to use GPU devices. A computer program that performs this type of operations would execute sequentially on a standard PC computer, while on the GPU device the data is transferred into the device memory and is divided on thousands of parallel-processing threads. Applications that operate with long arrays or matrices with large dimensions are appropriate for parallelization using this approach and in these cases a considerable speed-up of the calculations is achieved.

In this paper special software architecture called CUDA (abbreviated from Compute Unified Device Architecture) [19] was used for programming of GPU devices using upper level programming languages, like C and C++.CUDA is actually an extension of the programming language C, with a specific set of new keywords that are used for GPU commands. CUDA programming enables the source code to be executed on two different platforms, on a so-called host system (that represents the CPU) and a device system (that consists of one or more GPUs).

The task of programmer is to modify the existing program and to separate parts that perform a large number of calculations from the rest of the program and to transfer this data to the GPU device. The rest of the program is executed on the CPU. During the execution of the parallel version, the GPU works as the coprocessor of the CPU. Large data set on which the arithmetical calculations are performed is passed on to the GPU, while CPU interprets and transfers the necessary information.

In order to execute part of the program on a GPU device, the programmer must define specialized functions that are called kernels. Every kernel function is called from the main program and is executed on a grid of threads. Within every kernel call it is necessary to define the number of threads and dimension of the grid and of course to pass the appropriate arguments.

4.1 NVIDIA Tesla

Most of modern graphics cards, that are part of the configuration of modern personal computers, can be used as a GPU device. However, NVIDIA has also offered specialized GPU devices – the Tesla series [20]. Tesla devices are optimized for execution of complex scientific simulations. The main difference between a typical graphics card and a Tesla device is that Tesla does not have the possibility to output images to a display. For parallelization of LB solver, implementation and testing of improvements and execution time measurements Tesla C2075 device was used. This device is part of the Curie supercomputer, owned by GENCI, located in France and it represents PRACE (Partnership for Advanced Computing in Europe) Research Infrastructure resource. We were able to use it since our project was rewarded with access to this cluster within one of preparatory project access calls.

4.2 Parallelization of the software based on lattice Boltzmann method

In the implementation of lattice Boltzmann method the collision and propagation step (defined in Equations (5) and (6)) are repeated in a predefined number of iterations (or until a condition is satisfied). As it was already stated in section 2, each of these steps must be applied on all lattice nodes, before the next step starts. Since the equations that simulate collisions between particles are mutually independent, this step is a completely local operation. In the propagation step only the currently observed node and its closest neighbors are considered. All this indicates that this large data set can be divided to smaller subsets and then a certain number of (mostly arithmetical) operations should be performed on these subsets in parallel. Therefore it is evident that LB method is very suitable for parallelization. Figure 2 shows the algorithm of execution of standard (sequential) LB solver, with marked part of the algorithm that can be parallelized. In the same Figure parts of the source code used to implement the two mentioned steps are stated.

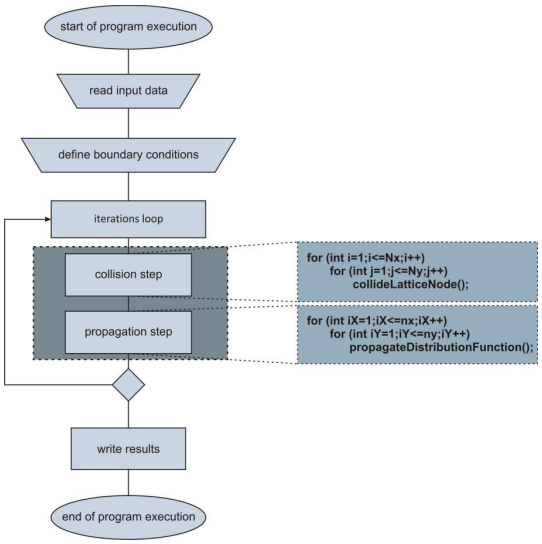

Figure 2 - Algorithm of execution of the standard LB solver

Since it was concluded that the collision and propagation steps are local in nature and that they can be easily parallelized, it is necessary to develop two kernel functions that will be called instead of regular functions used in standard version of LB solver. Figure 3 shows the calls of these functions in sequential (regular) and parallel form.

When two kernels required for collision and propagation step are implemented, they are called instead of the equivalent regular (sequential) functions. However, there is another important aspect that needs to be considered when existing programs are parallelized - the memory manipulation. Namely, GPU device is not a completely independent device, since the CPU controls the execution process on the GPU.CUDA architecture is designed in such a manner that CPU (also called host) and GPU (also called device) have separate DRAM (abbreviated from Dynamic Random Access Memory), which in fact is the case, if it is observed from hardware point of view. They are called host and device memory, respectively. CUDA allows the program to access these memories independently, but there are certain limitations in this process. Part of the program that is executed on the CPU has access only to the host memory, while kernel functions have access only to the device memory. Therefore it is necessary to explicitly allocate and free memory resources on both systems, as well as transfer data between these two types of memory. Special CUDA functions are providing the transfer of data from the CPU to the GPU and vice versa.

The easiest way to parallelize LB solver would be to save all the necessary data about lattice nodes in host memory. Before every kernel call this data would be transferred to device memory and after the kernel execution the new data would be transferred back to the host memory. But when transferring memory it is very important to analyze the effects that communication between CPU and GPU will have on the performance of parallelized applications. Thus the described approach is completely wrong, because the transfer of data between these two memories is a very “expensive” operation. Maximum bandwidth between device memory and the GPU is 102.4 GBps (Gigabytes per second) for Tesla C1060, while the bandwidth between host and device memory is around 8 GBps [21]. This means that it is necessary to minimize the transfers between host and device memory whenever it is possible. One should always balance between speed-ups obtained using GPU and the cost of data transfers between host and device.

In this concrete case, that means that all data about lattice nodes (the components of the distribution function and everything else) have to be initialized on the host, then this data is transferred only once to the device memory and during program execution they stay in device memory. Kernel functions access data in device memory and make the necessary changes. In order to view the results (visualize velocity and pressure field), only the macroscopic quantities can be transferred back to host memory. These macroscopic quantities are calculated in a specialized kernel function. It should be noted that fluid velocity and pressure fields are not visualized after every iteration, but most commonly after 100, 1000 or even more iterations, depending on the problem that is being simulated. This means that the amount of data transferred from host to device is significantly reduced.

There are some other aspects of memory manipulation that need to be considered when implementing kernel functions. There is a hierarchy in GPU device memory, i.e. threads can access data from memory at different levels. Every thread has its own private local memory and only that particular thread has access to it. Every block of threads has its own shared memory that is accessible to all threads within that block. And finally, all threads have access to global device memory. It does not take the same time for a thread to access data from local (register), shared or global memory. The least amount of time is needed to access register memory, more time is needed when accessing shared memory and the access to global memory requires the greatest amount of time [20; 22] (in some cases access to global memory is up to 150 times slower than access to register memory). Thus, access to global memory should be avoided whenever possible.

If all these facts are taken into consideration when performing parallelization of LB solver, instead of using two kernel functions for collision and propagation steps, it is more appropriate to implement one function that combines these two steps. That does not make any changes in the theoretical background, since the two steps are still performed separately. This is just a matter of implementation. Values of the distribution function

$f_i^{out}(\mathbf{x},t)$ in equations (5) and (6) (at the end of collision step and at the beginning of propagation step) are simply not written into the global device memory and then read again, but are written in register memory of each thread individually. In order to ensure that the collision step is finished before the start of the propagation for all nodes (i.e. for all threads in the kernel function), a system-defined CUDA function is used to synchronize thread execution - __syncthreads(). This function ensures that all threads are done with the execution of all previous operations before proceeding with kernel execution.

Figure 4 shows the algorithm of execution of parallelized LB solver.

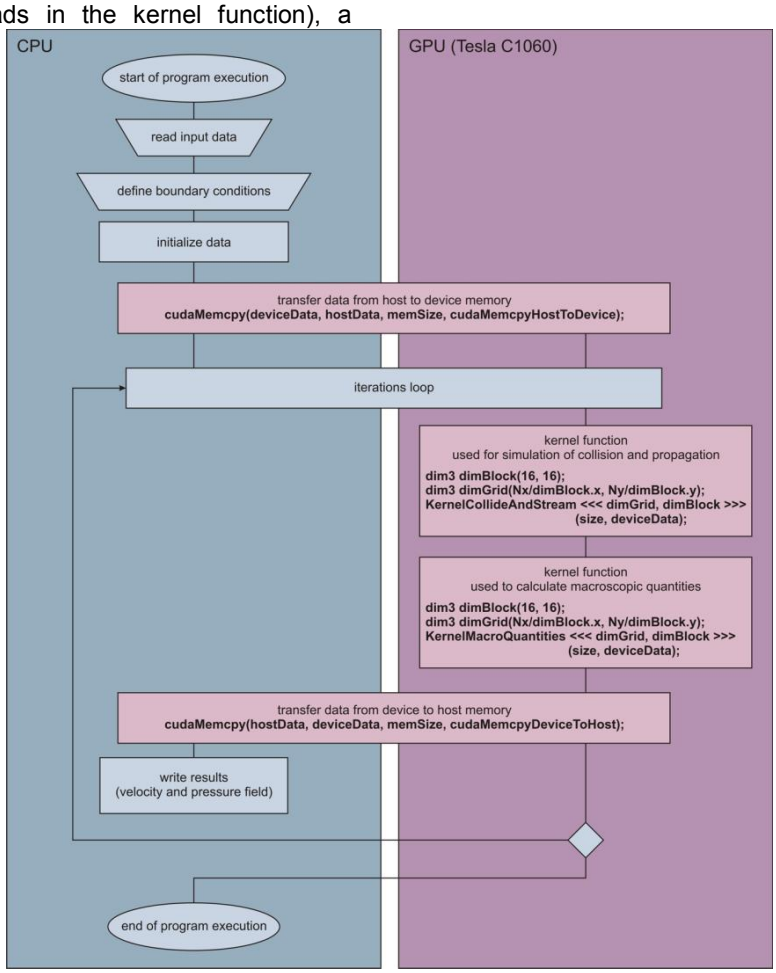

Figure 3 - Algorithm of execution of parallelized LB solver

5. *Results and Comparison of Execution Time of Regular and Parallelized Version of LB Solver*

In the sequel the speed-up obtained with the parallelized version of LB solver will be presented. First the importance of proper memory manipulation in CUDA programs will be demonstrated. As a test example the simulation of straight stationary flow between two parallel walls is used. The execution time of the program mainly depends on the dimension of the domain (the total number of nodes in lattice mesh) and on the number of iterations. In this case the performance tests have been performed using the total number of 20.000 nodes and the number of iterations is set to 20.000.The number of iterations is chosen such that the simulations can be finished in a reasonable amount of time and the obtained results are relevant because the computational load remains the same for every iteration and the insight that can be obtained using these test simulations can be used to estimate the time needed for more complex simulations.

The execution time of regular LB solver was compared with execution time of three different versions of parallelized LB solver. The first version transfers data from device to host memory and vice versa after every iteration (after every kernel call). The second version is optimized from the aspect of memory transfers – an additional kernel function that calculates macroscopic quantities is introduced and only macroscopic quantities are transferred back to host memory, to visualize the results. In the third version the access to global device memory is minimized, i.e. the collision and propagation steps are combined in one kernel function, like it was already described in Section 4.2.2. The comparison of execution time (expressed in seconds) is shown in Figure 4.

From Figure 4 it is obvious that bad memory manipulation can have a great influence on the

execution time. The first version of parallelized LB solver is executed almost 10 times slower than the sequential version, which is of course unacceptable. Proper memory manipulation (like in versions 2 and 3) has brought a speed-up of almost 10 times compared to the regular version, which is an expected speed-up. If the execution time of second and third version is compared, it can be seen that the third version is executed faster, due to a more optimal approach to global memory manipulation, but the difference is not as extreme as with the first version.

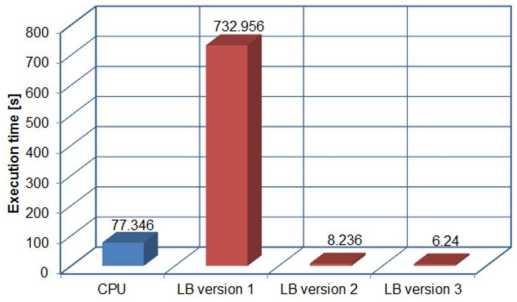

Figure 4 - Comparison of execution time of different versions of LB solver

The performance improvement of the existing program and the speed-up achieved by parallelization depend on many factors, but one of the main factors is the percentage of the program that can be parallelized. Amdahl's law specifies the maximum speed-up that can be expected by parallelizing portions of a program [22]. Maximum speed-up can be determined using the following expression:

$$S = \frac{1}{(1-P)+\frac{P}{N}} \qquad (7)$$

where P is the fraction of the program that can be parallelized (the easiest way to determine is by dividing the execution time of that particular portion with the overall execution time of the program), and is the number of processors.

In case of LB solver it can be considered that the number of processors is equal to the number of CUDA cores – 240. However, if the expression (7) is analyzed in detail, it becomes evident that as the number of processor increases, the ratio decreases and becomes a lower order member that can be neglected. Therefore with the increase of number of processors, this variable actually does not provide a greater performance benefit. It can be considered that this conclusion is valid in this particular case too.

Portion of the LB solver that is not parallelized includes the initialization of data and visualization of results. Therefore one can say that the portion of the program that is not parallelized is around 4.5% of the overall program, which means that .
If the values for and are substituted in the initial expression, it is obtained:

$$S \approx 21 \qquad (8)$$

Hence it is estimated that the maximum speed-up that can be achieved by parallelization of LB solver (and applying one Tesla C1060 GPU device) is 21.

When the execution time of the parallelized LB solver is compared with the regular solver, it is necessary to keep in mind the number of lattice nodes, i.e. the dimension of matrices on which the operations are performed. Figure 5 shows how the variation of speed-up of the parallelized LB solver is changing depending on the number of lattice nodes. As the number of nodes increases, the speed-up increases as well. Practically this means that the parallelized LB solver achieves greater speed-ups when simulation parameters are set such that the whole simulation is more computationally demanding. If the speed-up is determined based on the execution time of the entire program, the maximum value is 16.5 times (for 80.000 nodes). Further increase of the number of nodes does not affect the speed-up.

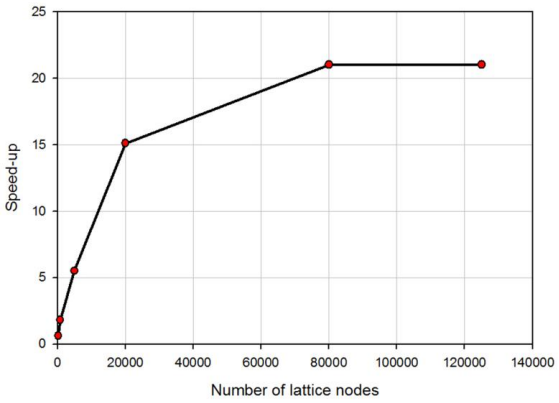

Figure 5 - Variation of the speed-up of parallelized LB solver depending on the number of lattice nodes

The achieved speed-up of 16,5 times is slightly lower than the predicted value of 18,5 times. This can be explained with the fact that even though they are minimized, the transfers of data between host and device memory do exist and a certain amount of time is required for their execution, and this time is not directly spent on operations that are normally performed in the regular LB solver. Figure 6 shows the values of speed-up calculated based on the execution time measured only for the portion of the program where calculations are performed (the iterations loop shown in Fig. 5, without visualization of results). In this case, the obtained speed-up is up to 21 times, which is approximately equal to the predicted value.

6. CONCLUSION

It seems that scientific simulations demand more and more computational resources. Personal computers can no longer follow up these increased demands. NVIDIA has offered a good solution for solving this emerging problem with its series of Tesla GPU devices. Applying GPU devices a considerable speed-up can be achieved in existing complex and demanding programs. Thanks to a specialized technology

that was developed by NVIDIA, the so-called CUDA architecture, customizing the existing programs to work with GPU devices is significantly simplified. In this paper the developed software based on lattice Boltzmann method was parallelized using CUDA architecture and the execution time of regular and parallel version was compared. On a specialized Tesla device the speed-up of 21 times was achieved, which represents a great time saving. It was shown that the developed parallel LB solver achieves greater speed-ups when simulations are more computationally demanding. It was also shown that the cost of CPU-GPU communication in memory manipulation requires special consideration from programmer's side, in order to obtain greater performance improvements.

REFERENCES

[1] Tirado-Ramos, A., Sloot, P.M.A., Hoekstra A.G., Bubak M., "An integrative approach to high-performance biomedical problem solving environments on the Grid," Parallel Computing, 2004, 30(9–10), pp. 1037–1055

[2] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P., "Brook for GPUs: stream computing on graphics hardware," ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH, 2004, 23(3), pp. 777-786

[3] Strzodka, R., Doggett, M., Kolb, A., "Scientific computation for simulations on programmable graphics hardware," Simulation Modeling Practice and Theory, 2005, 13(8), pp. 667–680.

[4] Anderson, J.A., Lorenz, C.D., Travesset, A., "General purpose molecular dynamics simulations fully implemented on graphics processing units," Journal of Computational Physics, 2008, 227(10), pp. 5342–5359.

[5] Vintache, D., Humbert, B., Brasse, D., "Iterative Reconstruction for Transmission Tomography on GPU Using Nvidia CUDA," Tsinghua Science & Technology, 2010, 15(1), pp. 11-16.

[6] Díaz, D., Esteban, F.J., Hernández, P., Caballero, J.A., Dorado, G., Gálvez, S., "Parallelizing and optimizing a bioinformatics pairwise sequence alignment algorithm for many-core architecture," Parallel Computing, 2011, 37, pp. 244-259.

[7] Kojic, M., Filipovic, N., Stojanovic, B., Kojic, N., "Computer modeling in bioengineering: Theoretical Background, Examples and Software," Chichester, England, John Wiley and Sons, 2008.

[8] Filipović, N., Kojić, M., Decuzzi, P., Ferrari, M., "Dissipative Particle Dynamics simulation of circular and elliptical particles motion in 2D laminar shear flow," Microfluidics and Nanofluidics, 2010, 10(5), pp. 1127-1134.

[9] Harada, T., Koshizuka, S., Kawaguchi, Y., "Smoothed Particle Hydrodynamics on GPUs," Proc. of Computer Graphics International, 2007, pp. 63-70

[10] Rossinelli, D., Bergdorf, M., Cottet, G.H., Koumoutsakos, P., "GPU accelerated simulations of bluff body flows using vortex particle methods," Journal of Computational Physics, 2010, 229(9), pp. 3316-3333

[11] Li, W., Zhe, Z., Wei, X., Kaufman, A., "GPU-Based flow simulation with complex boundaries," *GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, 2005, Chapter 47, pp. 747-764.

[12] Fan, Z., Qiu, F., Kaufman, A., Yoakum-Stover, S., "GPU Cluster for High Performance Computing," *ACM / IEEE Supercomputing Conference, 2004*.

[13] Xian, W., Takayuki, A., "Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster," Parallel Computing, 2011, 37(9), pp. 521-535.

[14] Wolfram, S., "Cellular Automaton Fluids 1: Basic Theory," J. Stat. Phys., 1986, 3/4, pp. 471–526.

[15] Đukić, T, "Modeling solid-fluid interaction using LB method," Master's thesis, Faculty of Engineering, Kragujevac, 2012.

[16] Malaspinas, O.P., "Lattice Boltzmann Method for the Simulation of Viscoelastic Fluid Flows," PhD dissertation, Switzerland, 2009.

[17] Bhatnagar, P.L., Gross, E.P., Krook, M., "A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems," Phys. Rev. E, 1954, 77(5), pp. 511-525.

[18] Sukop, M.C., Thorne, D.T. Jr., "Lattice Boltzmann Modeling - An Introduction for Geoscientists and Engineers," Heidelberg: Springer, 2006.

[19] NVIDIA,"CUDA Programming guide, Version 4.0," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2011, May.

[20] NVIDIA Corporation, "Tesla Personal Supercomputer," http://www.nvidia.com/object/personal-supercomputing.html, 2013, March.

[21] NVIDIA, "CUDA C Best practices guide, Version 4.0," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf, 2011, May.

[22] Amdahl, G., "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *AFIPS Conference Proceedings, 1967, 30*, pp. 483–485.