Implementation of the Lattice Boltzmann Method on Heterogeneous Hardware and Platforms using OpenCL

Predrag M. TEKIĆ, Jelena B. RADJENOVIĆ, Miloš RACKOVIĆ
Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 3, 21000 Novi Sad, Serbia
tekic@uns.ac.rs, radjenovic@uns.ac.rs, rackovic@dmi.uns.ac.rs

***Abstract*—The Lattice Boltzmann method (LBM) has become an alternative method for computational fluid dynamics with a wide range of applications. Besides its numerical stability and accuracy, one of the major advantages of LBM is its relatively easy parallelization and, hence, it is especially well fitted to many-core hardware as graphics processing units (GPU). The majority of work concerning LBM implementation on GPU's has used the CUDA programming model, supported exclusively by NVIDIA. Recently, the open standard for parallel programming of heterogeneous systems (OpenCL) has been introduced. OpenCL standard matures and is supported on processors from most vendors. In this paper, we make use of the OpenCL framework for the lattice Boltzmann method simulation, using hardware accelerators – AMD ATI Radeon GPU, AMD Dual-Core CPU and NVIDIA GeForce GPU's. Application has been developed using a combination of Java and OpenCL programming languages. Java bindings for OpenCL have been utilized. This approach offers the benefits of hardware and operating system independence, as well as speeding up of lattice Boltzmann algorithm. It has been showed that the developed lattice Boltzmann source code can be executed without modification on all of the used hardware accelerators. Performance results have been presented and compared for the hardware accelerators that have been utilized.**

***Index Terms*—GPU, Java, lattice Boltzmann method, many-core, OpenCL.**

I. Introduction

In recent years there has been an astounding rise in numerical computer simulations in nearly every area of science and engineering. Rapid improvements in CPU performance (Moore's Law) and price drops caused by mass production of computer hardware and intense competition between leading computer vendors, led to development of related areas and among them to numerical computer simulations. Another side effect of hardware mass production was development of cluster computing. Clusters where comprised of disused computing units (commodity desktop computers), and achieved the same level of performance, or even outperformed traditional high performance computers at a fraction of the price.

However, a few years ago the computer processors industry hit a serious frequency wall, implying that increasing the processor's clock-rate for gains in performance could not be done indefinitely, due to increases in power consumption and heat generation (power wall). Processor frequency rates have been stabilized at around 3 GHz. That was the turning point in the computer processor industry, all the major processor vendors started manufacturing multi-core processors and all the major GPU vendors turned to many-core GPU design. Processor manufacturers continued to increase the power of their products, without raising the processor frequency barrier. Multi-core processor is single component with two or more independent processors (cores), and many-core processor is one with the number of cores large enough that traditional multi-processor techniques are no longer efficient (several tens of cores with fraction of the power of the CPU's).

There is a considerable cost associated with software development for GPUs: their architecture is quite different from that of a conventional computer and code must be (re)written to explicitly expose algorithmic parallelism. Various GPU programming models have been established, which are usually unfamiliar and vendor specific, and often require advanced knowledge of their hardware design. Low-level, device-specific assembly languages e.g. Compute Abstraction Layer for AMD GPUs to high-level software development kits such as CUDA SDK for NVIDIA's GPUs; ATI Accelerated Parallel Processing (APP) SDK for ATI's GPUs; IBM Cell SDK for the Cell BE processors, while traditional multi-core processors (Intel, AMD) typically involve OpenMP directive extensions for C and Fortran. The most popular and most mature development tool for scientific GPU computing, among quoted, has proved to be CUDA (Compute Unified Device Architecture). It has been invented by the vendor NVIDIA for its GPU products. It defines a C dialect for writing scalar GPU programs along with a set of C language extensions for simplifying the GPU control from the host program. Despite its popularity in the community, it is a proprietary vendor-controlled language tied to NVIDIA hardware.

Over the past couple of years an effort has been made to unify the parallel software development for all different computer architectures under one standard – the Open Computing Language (OpenCL). OpenCL is an open, royalty-free standard for cross-platform, parallel programming of heterogeneous processors announced by the Khronos Group. OpenCL has attracted vendor support, with implementations available from NVIDIA, AMD, Apple and IBM. The standard comprises of an abstract model for the architecture and memory hierarchy of OpenCL-compliant compute devices, a C programming language for compute device code and a host-side C API. Because the standard has been designed to reflect the design of contemporary

Digital Object Identifier 10.4316/AECE.2012.01009

1582-7445 © 2012 AECE

hardware there are a lot of similarities with the CUDA programming model. The archetypal OpenCL platform consists of a host computer to which one or more compute devices are attached, each of which is in turn comprised of one or more compute units, each having one or more processing elements. The execution model for OpenCL consists of the controlling host program and kernels which execute on OpenCL devices. To scientific programmers, the OpenCL standard may be an attractive alternative to CUDA, as it offers a similar programming model with the prospect of hardware and vendor independence. To the best of the author's knowledge, only two papers have been published concerning OpenCL [1],[2].

Recently, the lattice Boltzmann method (LBM) has become an alternative method for computational fluid dynamics (CFD) [3]-[6]. Unlike conventional methods, which are based on macroscopic continuity equations, it utilizes the particle distribution function to describe collective behavior of fluid molecules. The kinetic features of the LBM enable it to be a very effective numerical tool for simulating complex geometries flows, multiphase flows, magnetohydrodynamic systems, etc. LBM is very suitable for parallel computation due to the local property of the dominant equations. Its simple kernel structure qualifies the method for high-performance computing. [7]-[12] Parallel implementation of the LB method using GPU architecture has gained remarkable attention in recent years. Utilization of GPU's for high-performance computing can demonstrate significant performance benefits, and relative to CPU implementations, GPU implementations of the lattice Boltzmann method often achieve performance increases of an order of magnitude. Li at al [13] attained first promising results of a LBM based flow simulation on GPU's. More recently, Tolke and Krafczyk[14] implemented D3Q19 lattice Boltzmann kernel on NVIDIA GPU's using CUDA. Kuznik et al. [15] provide and implementation of a general purpose LBM code where all steps of the algorithm are running on the GPU. Habich et al. [16] presented performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on NVIDIA GPU's using CUDA. A new approach to the LBM for graphics processing units and some optimization principles for CUDA programming are presented by Obrecht et al.[17]. Xian and Takayuki [18] were executed successfully the LBM on multi-node GPU cluster by using CUDA and MPI library.

The main objective of the present work is to implement the lattice Boltzmann method according to OpenCL specification. The problem of flow in the lid-driven cavity, has been used for the study of this programming approach. Software application that has been developed to simulate lid-driven flow, in deep cavities, comprises of a host program and kernels. Controlling (host) program has been developed using one of the mainstream programming languages, Java. Kernels were written to parallelize performance of the most intensive parts of the lattice Boltzmann method algorithm, in accordance with OpenCL specification. Java library (JOCL) [19] has been used as a binding between host (Java) and kernel (OpenCL) programs. Software application has been tested on GPU's from different vendors, both NVIDIA and AMD. Results of the simulations have been presented. Platform and hardware independence have been accomplished by this combination of programming languages.

The rest of this paper is organized as follows: Section 2 and 3 present the mathematical formulation of the lattice Boltzmann method and the implementation of lid-driven deep cavity flow model using Java and OpenCL framework; Section 4 presents the results of our study on the performance of the implemented model using different vendor hardware; Summary and conclusions are given in section 5.

II. METHODS

A. Lattice Boltzmann equation

In the following section a brief introduction of the lattice Boltzmann method is given. More detailed description can be found elsewhere [3]-[6],[20],[21].

The lattice Boltzmann equation (LBE) reads:

$$\frac{\partial f}{\partial t} + \xi \cdot \Delta f = \Omega. \tag{1}$$

After applying simple Bhatnagar-Gross-Krook (BGK) approximation on the collision term (Ω), a space discretized LBE is as follows:

$$f_i(x + e_i\Delta t, t + \Delta t) - f_i(x,t) = -\frac{1}{\tau}\left[f_i(x,t) - f_i^{eq}(x,t)\right] \tag{2}$$

Where $f_i(x,t)$ and $f_i^{eq}(x,t)$ are the distribution function and the equilibrium distribution function, respectively, e_i is a discrete velocity vector and τ is the single relaxation time related to the kinematic viscosity of the fluid:

$$\tau = \frac{6\nu + 1}{2}. \tag{3}$$

The nine-velocity square lattice model (D2Q9) is commonly used for simulating two-dimensional (2-D) flows. The equilibrium distribution function for isothermal incompressible flow and D2Q9 model reads:

$$f_i^{eq} = \rho w_i\left[1 + \frac{3}{c^2}e_i \cdot u + \frac{9}{2c^4}(e \cdot u)^2 - \frac{3}{2c^2}u \cdot u\right] \tag{4}$$

where W_i and u are weight parameters and velocity of the fluid, respectively. Discrete velocity vectors are defined as:

$$\begin{aligned} &e_0 = (0,0) \\ &e_i = (\pm c,0),(0,\pm c) \qquad i = 1,2,3,4 \\ &e_i = (\pm c,\pm c) \qquad i = 5,6,7,8 \end{aligned} \tag{5}$$

where c=Δx/Δt, Δx and Δt are the lattice grid spacing and the time step, respectively. Weight parameters for each velocity vector are $W_0 = 4/9$, $W_{1,3,5,7} = 1/9$ and $W_{2,4,6,8}=1/36$. The macroscopic density and velocity of the fluid are obtained from the distribution function as follows:

$$\begin{aligned} \rho &= \sum_{i=1}^{8} f_i \\ \rho u &= \sum_{i=1}^{8} e_i f_i. \end{aligned} \tag{6}$$

In this work, the simulation of lid-driven flow in two-dimensional deep cavity has been implemented and used as a benchmark problem. The code and simulation results have been validated according to the study of Patil et al. [21], where the simulation details, initial and boundary conditions used can also be found. Performance results are presented for the deep cavity with aspect ratio of 1.5 and Reynolds number of 1000.

B. OpenCL

Open Computing Language (OpenCL) was initially proposed by the Apple Corporation to the Khronos Group. A working group was formed that included representatives from GPU, CPU and software companies. Results of their five month work were technical details for the specification OpenCL 1.0. As the standard matured it became supported on the processors of most vendors (NVIDIA, AMD, Apple, IBM, Intel). Major GPU vendors included OpenCL specification support in software development kits for most of their products. OpenCL specification and performance of its software implementations on different vendor GPU's have been taken under examination in this work.

Higher utilization of parallelism, available in contemporary processors, is provided by OpenCL standard, and is still a feasible learning curve for programmers familiar (abreast) with C programming language. Necessary changes have been made to the C programming language in order to allow parallel computing on all the different processor architectures. Scientific computing community requirements related to the numerical precision (which should be consistent across the different hardware and vendors) were incorporated in OpenCL standard. Matter that is of great importance to the scientific computing community is numerical precision. This matter is incorporated in OpenCL standard in order to provide mathematical consistency across the different hardware and vendors.

The prototype OpenCL platform consists of a host computer to which one or more compute devices are attached. A Computing device is comprised of one or more compute units, each having one or more processing elements. The execution model for OpenCL consists of the controlling host program which is executed on the host, and kernels which are executed on one or more OpenCL devices. Kernels are executed as multiple instances called work-items which are grouped into work-groups. The OpenCL memory model is divided into four distinct memory regions: each computing device has a pool of global memory to which all work-items in all work-groups may read and write; local memory which is local to a work-group; private memory which is private to each work-item; and constant memory, a read-only region of global memory which is allocated and is initialized by the host program. OpenCL specification defines these regions only in terms of their access properties; the relative speeds and physical location of these memories is strictly implementation-specific.

Parts of code containing performance intensive routines should be (re)written by computational scientists as OpenCL kernels, in order to be executed on computational hardware. Host programs should be written according to OpenCL API, which gives functions needed from locating computing enabled hardware connected to the system for compiling, submitting, queuing and synchronizing the compute kernels on the hardware. Kernel execution and management of data transfers, between host and computational hardware is done by the OpenCL runtime.

First step in the OpenCL host program is the creation of context, where all the operations will be performed. It is possible that the context has more than one associated device (CPU,GPU), and if that is the case, between those devices relaxed memory consistency is guaranteed by the OpenCL standard. Buffers (1-dimensional blocks of memory) and images (2- and 3-dimensional blocks of memory) are used by the OpenCL to store the data of the kernel that is to be run on the specified compute device. In order to run kernel on the specified compute device, memory for the kernel data needs to be allocated and kernel needs to be loaded and built. Kernel object should be built and kernel arguments should be set in order to call a kernel. Command queue must be created for all the computations that will be executed on the selected compute device. Created command queue has one-to-one mapping with the device. After the creation of a command queue, the kernel can be queued for the execution. Global work is the total number of the elements (indexes) in the launch domain, and individual elements are work items. The kernel is executed in parallel on a 1-, 2- or 3-dimensional domain of indexes.

C. Java and OpenCL

Java has established itself in last decade and a half, as one of the mainstream programming languages. It is object oriented programming language designed to have as few implementation dependencies as possible, allowing application developers portability of the written code. Because of its characteristics, it has also gained popularity in the scientific community.

Programming language that has been chosen for the host program provides portability of the developed code. Application comprised of the quoted technologies has the benefits of taking advantage of available parallel architectures through the use of OpenCL specification, in solving demanding fluid simulations. Also, it has great platform portability and hardware independence as a consequence of the nature of the selected technologies, Java and OpenCL, respectively.

For the purpose of this experiment, an open source library for binding Java and OpenCL specification functionalities has been employed. The library that has been utilized (JOCL) provides Java-bindings very similar to original OpenCL API. Functions are provided as the static methods. OpenCL implementations in the available software development kits (SDK's) are packed inside the Dynamic-link library (OpenCL.dll) and installed on the operating system. Because of this, purpose of the library (JOCL) that has been used is to communicate with this library (OpenCL.dll) employing Java Native Interface (JNI). Platform that has been utilized for this simulation was 64 bit, Windows 7 Ultimate edition, along with the appropriate graphics card drivers.

In this work we will be discussing two of the currently available implementations. The NVIDIA GPU Computing SDK, version 3.1.1, with support for OpenCL 1.0, and the the AMD APP (formerly ATI Stream SDK), version 2.3, with support for OpenCL 1.1. There are also two more implementations available from Apple (OpenCL in the Apple Snow Leopard) and Intel (Intel OpenCL SDK) which have not been taken under consideration in this work.

III. IMPLEMENTATION

In the following code samples OpenCL source code has been presented. Most computationally intensive parts of the lattice Boltzmann method, streaming and collision, have been rewritten according to OpenCL specification.

The first step in the initialization of the device that will be used for execution of the OpenCL code is to obtain platform ID. It is possible that one host has one or more platforms attached. After that, we need to initialize context properties.

cl_platform_id platforms[] = new cl_platform_id[1];
clGetPlatformIDs(platforms.length, platforms, null);
cl_context_properties contextProperties = new cl_context_properties();
contextProperties.addProperty(CL_CONTEXT_PLATFORM, platforms[0]);

Also, after the initialization of the context properties, we need to create the context on GPU device that we will be using.

cl_context context = clCreateContextFromType(contextProperties, CL_DEVICE_TYPE_GPU, null, null, null);
Get the list of GPU devices associated with the context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, null, numBytes);

Obtaining ID of the first device associated with the context

int numDevices = (int) numBytes[0] / Sizeof.cl_device_id;
cl_device_id devices[] = new cl_device_id[numDevices];
clGetContextInfo(context, CL_CONTEXT_DEVICES, numBytes[0], Pointer.to(devices), null);

In the following line we create command queue on the selected device

cl_command_queue commandQueue = clCreateCommandQueue(context, devices[0], 0, null);
Allocation of the memory object for input and output data
cl_mem memObjects[] = new cl_mem[23];
memObjects[0] =
*clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, Sizeof.cl_int*2, src_host, null);*

In the following section of the code we create a program from source, at runtime, and build it

cl_program program = clCreateProgramWithSource(context, 1, new String[]{ Source }, null, null);
clBuildProgram(program, 0, null, null, null, null);

In the following section we create a kernel from program and start with setting the arguments which are pointers to the memory objects created on the host.

cl_kernel kernelCollisionProp = clCreateKernel(program, "CollisionPropogate", null);
clSetKernelArg(kernelCollisionProp, 0, Sizeof.cl_mem, Pointer.to(memObjects[0]));

We add kernel to command queue for the execution on the selected OpenCL device

clEnqueueNDRangeKernel(commandQueue, kernelCollisionProp, 2, null, global, local, 0, null, null);
clFinish(commandQueue);

At this point of the program control is at the device that executes a kernel (kernelCollisionProp) in a data parallel mode (performing the same task on different data). Three kernels have been created: kernelCollisionProp, kernelStreamingProp and kernelBoundaryProp. In the figure 1 is represented lattice Boltzmann time step, every block is realized as one kernel procedure. After the execution of the kernels, program returns control to host which needs to release resources on the device.

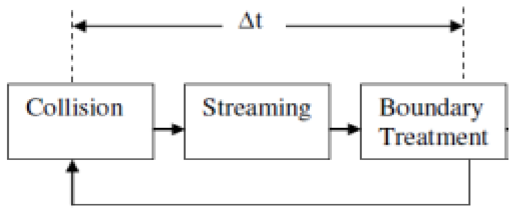

Figure 1. Lattice Boltzmann time step representation.

We are retrieving the data from the device to host

*clEnqueueReadBuffer(commandQueue, memObjects[20], CL_TRUE, 0, nx*ny * Sizeof.cl_float, src_host_u, 0, null, null);*

When the program execution is finished, we need to release resources (memory object, kernel, program, command queue and context).

clReleaseMemObject(memObjects[0]);
clReleaseKernel(kernelCollisionProp);
clReleaseProgram(program);
clReleaseCommandQueue(commandQueue);
clReleaseContext(context);

Parallel code for simulation of lid-driven flow in two-dimensional deep cavities that has been developed in this work has the benefits of the hardware and platform independence. It can run on both platforms, which is a non-trivial benefit – it promises tremendous savings in parallel code development and optimization efforts. Before OpenCL standard, every platform would require implementations according to their own technologies, CUDA in the case of NVIDIA and AMD APP (ATI Stream) in the case of AMD ATI.

IV. RESULTS AND DISCUSSION

Listed in Table 1 are platform and device information obtained using the application written in Java, leveraging the JOCL library and OpenCL API functions. In the first column are parameter names, and in the following columns are written values obtained from the selected devices for the demanded parameters. Existing software development kits from AMD and NVIDIA have support for OpenCL specification. Platform from NVIDIA, NVIDIA CUDA, currently supports OpenCL specification version 1.0, and contemporary platform from AMD, ATI-Stream (now AMD APP), has support for OpenCL specification version 1.1. Simulation model, lid-driven flow in deep cavities using lattice Boltzmann method, has been implemented in accordance with OpenCL specification 1.0. The new, OpenCL 1.1, specification is backward compatible, and developed simulation model was run without any modifications on both platforms. Simulation was carried out on four different devices, two from each platform.

Device characteristics have been given in Table 2. Listed device characteristics are in direct connection with performance of the simulation. Number of computing units will prove to have the greatest influence on the performance of the simulation presented in this paper.

TABLE I. TESTING PLATFORMS DETAILS

Vendor	AMD		NVIDIA	
CL_DEVICE_NAME	Redwood Radeon HD 5570	AMD Athlon™ 7750 Dual-Core Processor	GeForce GT 220	GeForce 9800 GT
CL_PLATFORM_NAME	ATI Stream		NVIDIA CUDA	
CL_PLATFORM_VENDOR	Advanced Micro Devices, Inc.		NVIDIA Corporation	
CL_DEVICE_VENDOR	Advanced Micro Devices, Inc.	AuthenticAMD	NVIDIA Corporation	
CL_DEVICE_TYPE	GPU	CPU	GPU	
CL_DEVICE_OPENCL_C_VERSION	OpenCL C 1.1		OpenCL C 1.0	
CL_PLATFORM_VERSION	OpenCL 1.1 ATI-Stream-v2.3 (451)		OpenCL 1.0 CUDA 3.2.1	
CL_DEVICE_VERSION	OpenCL 1.1 ATI-Stream-v2.3 (451)		OpenCL 1.0 CUDA	
CL_DRIVER_VERSION	CAL 1.4.900	2	258.96	

TABLE 2. COMPUTE DEVICES CHARACTERISTICS

CL_DEVICE_NAME	GeForce 9800 GT	GeForce GT 220	Redwood Radeon HD 5570	AMD Athlon(tm) 7750 Dual-Core Processor
CL_DEVICE_GLOBAL_MEM_SIZE	1 054 408 704	1 034 485 760	536 870 912	3 221 225 472
CL_DEVICE_LOCAL_MEM_SIZE	16384	16384	32768	32768
CL_DEVICE_MAX_WORK_ITEM_SIZES	512 512 64	512 512 64	256 256 256	1024 1024 1024
CL_DEVICE_MAX_CLOCK_FREQUENCY	1375	1360	650	2712
CL_DEVICE_ADDRESS_BITS	32	32	32	64
CL_DEVICE_MAX_COMPUTE_UNITS	14	6	5	2

In order to confirm that the developed code can run without any modification requirements on GPUs manufactured by different vendors and different software development kits (SDK), two devices from different vendors that fall in the same price range were chosen for this test: NVIDIA's GeForce GT 220 and AMD ATI's Radeon HD 5570. The devices were paired up with an AMD Dual-Core CPU to demonstrate performance gains when executing developed code on GPU units over CPU units.

It has been concluded that the number of streaming multiprocessors has the highest impact on the time required to perform the simulation (the more streaming multiprocessors - the quicker the simulation will be completed), which has been determined by installing a more powerful GPU into the system - a GeForce 9800GT.

A. Performance results

The simulation performance results of the developed Java-OpenCL application have been presented in this section. As mentioned previously, simulation of lid-driven flow in two-dimensional deep cavity using lattice Boltzmann method has been implemented as a benchmark problem. Aspect ratio of 1.5 has been used, as well as Reynolds number of 1000. The grid resolution used for this model ranges from 130 x 195 (25,350 nodes) to 500 x 750 (93,750 nodes). Steady-state of the simulation is achieved after approximately 150 000 time steps (iterations). Presented performance results (execution times) are times required for completing 150 000 steps on the used hardware units. The simulation results have not been taken under consideration in this work, only the performance of the simulation on the selected hardware devices. The developed code and simulation results have been validated by comparison to the simulation results according to the study of Patil et al.[18].

GPU units from a major vendor have been used for the code performance evaluation. There were three GPU units and one CPU unit selected. NVIDIA GeForce models, GT 220 and 9800 GT, both with CUDA support and one AMD ATI model, Radeon HD 5570. AMD CPU unit has been used to compare performance gains, while using same (parallel) code, on GPU versus CPU units.

TABLE 3. CODE EXECUTION TIMES (MILLISECONDS) ON DIFFERENT HARDWARE UNITS.

Mash size GPU/CPU	130 x 195	200 x 300	250 x 375	500 x 750
NVIDIA 9800 GT	902175	2064689	3205952	13774814
NVIDIA GT 220	2761402	6595419	9578655	41103367
AMD ATI Radeon HD 5570 (Redwood)	3565055	8651798	12977574	54748548
AMD Athlon 7750 Dual-Core Processor	12142176	28345076	44238400	208901314

In Table 3. are displayed times (in milliseconds) required for completion of simulation on different hardware units. Results range from approximately quarter of an hour required for simulation on the smallest mesh size (130 x 195) and executed on Nvidia 9800 GT graphics processing unit, to 58 hours that will take AMD Athlon Dual-Core processor to finish simulation on the largest mesh size (500 x 750).

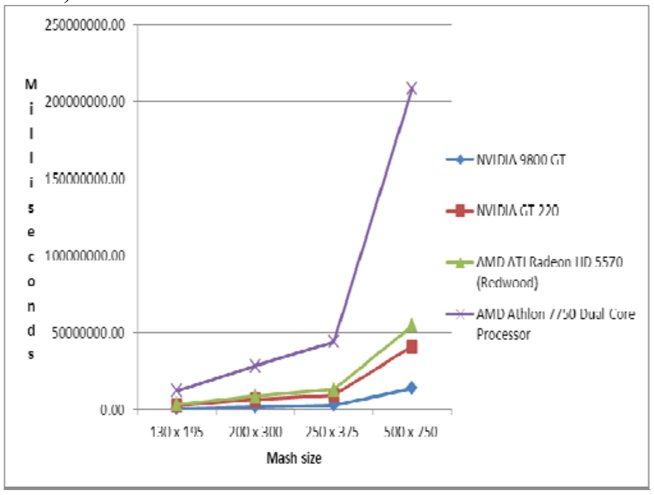

Figure 2. Graphic representation of simulation execution times.

Previous figure 2 represented graphical interpretation of the results given in previous table (Table 3.). From this figure the conclusion we can draw is that NVIDIA GPU's proved to have better performance then AMD ATI GPU. NVIDIA platform has better performance implementation of OpenCL specification, in spite of the fact that AMD platform has newer OpenCL specification implemented (OpenCL 1.1).

V. CONCLUSION

Platform and hardware independent source code has been developed for the fluid flow simulation using the lattice Boltzmann method. Modern framework for scientific computation (OpenCL) has been analyzed. It has been shown that this framework can be valuable for computational scientists, since it is hardware and vendor neutral, and still provides considerable performance improvements.

A popular programming language, Java, has been used in combination with OpenCL, as the host programming language. It demonstrates the ability of programming GPU's from Java and the possibility to exploit the computational power of GPU's for existing applications developed in Java.

Developed code has been executed without modification, on different hardware from different vendors for the purpose of this demonstration. Hardware accelerators, AMD ATI Radeon GPU, AMD Dual-Core CPU and NVIDIA GeForce GPU's, have been used. Performance of the utilized hardware has been evaluated. NVIDIA hardware showed better performance results, possibly because OpenCL framework has been developed according to NVIDIA hardware architecture and after CUDA programming framework. Our results suggest that an OpenCL-based implementation of the lattice Boltzmann method provides considerable performance improvements and yet maintains vendor and hardware autonomy.

The obtained simulation results have shown to be in good agreement with the results available in the literature.

ACKNOWLEDGMENT

Research was partially supported by the Ministry of Science and Technological Development of Republic of Serbia, through project no. OI-174023 'Intelligent techniques and their integration into wide-spectrum decision support'.

REFERENCES

[1] G. Khanna and J. McKennon, "Numerical modeling of gravitational wave sources accelerated by OpenCL," Computer Physics Communications, vol. 181 pp. 1605–1611, 2010.
[2] M. J. Harvey and G. D. Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," Computer Physics Communications, ARTICLE IN PRESS.
[3] S. Succi, The Lattice Boltzman Equation for Fluid Dynamics and Beyond. Oxford: Oxford University Press, 2001.
[4] D. Yu, R. Mei, L.-S. Luo, and W. Shyy, "Viscous flow computations with the method of lattice Boltzmann equation," Progress in Aerospace Sciences, vol. 39, pp. 329-367, 2003.
[5] L.-S. Luo, "The lattice-gas and lattice Boltzmann methods: Past, present, and future," in Proc Int Conf Appl Comput Fluid Dyn, Beijing, 2000, pp. 52-83.
[6] M. C. Sukop and D. T. J. Thorne, Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers. Berlin: Springer, 2007.
[7] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. A. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in IEEE International Symposium on Parallel and Distributed Processing, 2008, pp. 1-14.
[8] T. Pohl, et al., "Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures," presented at the Proceedings of the 2004 ACM/IEEE conference on Supercomputing, 2004.
[9] G. Wellein, T. Zeiser, G. Hager, and S. Donath, "On the single processor performance of simple lattice Boltzmann kernels," Computers & Fluids, vol. 35, pp. 910-919.
[10] D. Vidal, R. Roy, and F. Bertrand, "A parallel workload balanced and memory efficient lattice-Boltzmann algorithm," Computers & Fluids, vol. 39, pp. 1411–1423, 2010.
[11] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras, "A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries," Concurr. Comput. : Pract. Exper., vol. 22, pp. 1-14, 2010.
[12] K. R. Tubbs and F. T. C. Tsai, "GPU accelerated lattice Boltzmann model for shallow water flow and mass transport," International Journal for Numerical Methods in Engineering, vol. 86, pp. 316-334, 2011.
[13] W. Li, X. Wei, and A. Kaufman, "Implementing Lattice Boltzmann Computation on Graphics Hardware," Visual Computer, vol. 19, pp. 444-456, 2003.
[14] J. Tolke and M. Krafczyk, "TeraFLOP computing on a desktop PC with GPUs for 3D CFD," International Journal of Computational Fluid Dynamics, vol. 22, pp. 443-456, 2008.
[15] F. Kuznik, C. Obrecht, G. Rusaouen, and J.-J. Roux, "LBM based flow simulation using GPU computing processor," Computers & Mathematics with Applications, vol. 59, pp. 2380-2392, 2010.
[16] J. Habich, T. Zeiser, G. Hager, and G. Wellein, "Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA," Advances in Engineering Software, vol. 42, pp. 266-272, 2011.
[17] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, "A new approach to the lattice Boltzmann method for graphics processing units," Computers & Mathematics with Applications, vol. In Press, Corrected Proof.
[18] W. Xian and A. Takayuki, "Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster," Parallel Computing, vol. In Press, Corrected Proof.
[19] JOCL Library, http://www.jocl.org/
[20] P. M. Tekić, J. B. Rađenović, N. L. Lukić, and S. S. Popović, "Lattice Boltzmann simulation of two-sided lid-driven flow in a staggered cavity," International Journal of Computational Fluid Dynamics, vol. 24, pp. 383-390, 2010.
[21] D. V. Patil, K. N. Lakshmisha, and B. Rogg, "Lattice Boltzmann simulation of lid-driven flow in deep cavities," Computers & Fluids, vol. 35, pp. 1116-1125, 2006.