Performance comparison of Lattice Boltzmann fluid flow simulation using OpenCL and CUDA frameworks

Jelena Tekić, Predrag Tekić, Miloš Racković
University of Novi Sad, Faculty of Sciences, Novi Sad, Serbia
{radjenovic, tekic} @uns.ac.rs, rackovic@dmi.uns.ac.rs

***Abstract*—This paper presents performance comparison, of the lid-driven cavity flow simulation, with Lattice Boltzmann method, example, between CUDA and OpenCL parallel programming frameworks. CUDA is parallel programming model developed by NVIDIA for leveraging computing capabilities of their products. OpenCL is an open, royalty free, standard developed by Khronos group for parallel programming of heterogeneous devices (CPU's, GPU's, …) from different vendors. OpenCL promises portability of the developed code between heterogeneous devices, but portability has performance penalty. We investigate performance downside of portable OpenCL code comparing to similar CUDA code run on the NVIDIA graphic cards. Lid-driven cavity flow benchmark code, for both examples, has been written in Java programming language, and uses open source libraries to communicate with OpenCL and CUDA. Results of simulations for different grid sizes (from 128 to 896) have been presented and analyzed. Simulations have been carried out on an NVIDIA GeForce GT 220 GPU.**

Keywords: CUDA, OpenCL, Lattice Boltzmann, Java, GPU

I. INTRODUCTION

In recent years multi-core and many-core processors are replacing single core processors, especially Graphics Processing Units (GPUs) have greatly outperformed CPUs in memory bandwidth (Figure 1) and number of arithmetic operations per second. GPUs have an important role in today's high performance computing applications. GPUs brought high performance computing, which was privilege to small group of people, scientists, and reserved for large computer clusters, to every commodity desktop/personal computer.

Due to large processing power potential of GPUs, researchers and developers are becoming increasingly interested in exploiting this power for general purpose computing.

Specific scientific fields, like computational fluid dynamics (CFD), benefited from this trend, of increasing processing power of GPUs. Algorithms that can be relatively easily parallelized, like Lattice Boltzmann method, gain more popularity.

In this paper we investigate the performance differences between CUDA and OpenCL implementations of well-known CFD benchmark problem, one sided lid driven cavity flow. Code was developed using Java programming language, and open source java bindings libraries for OpenCL and CUDA, JOCL[4] and JCUDA[5].

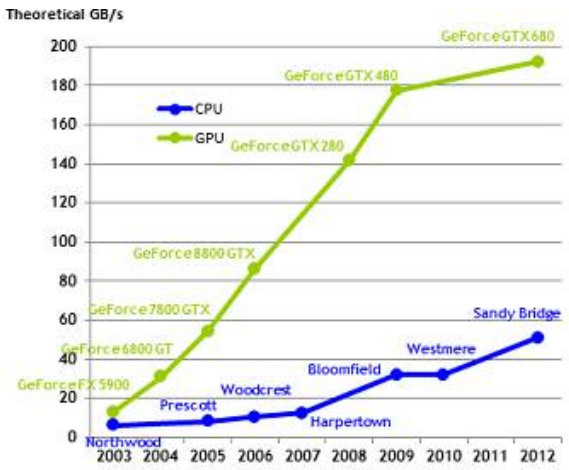

Figure 1. Memory bandwidth for the CPU - GPU

A. CUDA

Compute Unified Device Architecture (CUDA) [1] has been introduced by NVIDIA in 2006., as proprietary, vendor specific, API and set of language extensions for programming NVIDIA products. Considering that CUDA has been developed by the same company that produces hardware devices, it would be expected for CUDA code to perform better on their hardware products. Since, CUDA was another new device specific API and language, developers were forced to learn in order to utilize NVIDIA products, and that fact caused rise in demand for a single language and API that would be capable of dealing with any device architecture.

CUDA provides two different APIs, the *Runtime API* and *Driver API*. Both *APIs* are very similar regarding basic tasks like memory handling, and starting with CUDA 3.0 *APIs* are interoperable and can be mixed to some level. The most important differences between these two *APIs* is how *kernel's* are managed and executed.

B. OpenCL

Open Computing Language (OpenCL) [2] is an open, royalty free, standard developed by Khronos group [3] for parallel programming of heterogeneous devices (CPUs, GPUs, DSPs) from different vendors. OpenCL has attracted vendor support, with implementations available from NVIDIA, AMD, Apple and IBM. It was introduced in late 2008. Because the standard has been designed to

reflect the design of contemporary hardware there are a lot of similarities with the CUDA programming model.

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.

OpenCL code (kernel) can be compiled at runtime, which is not a case with CUDA compile model, and that add up to OpenCL execution time. On the other hand, this just in time compile model allows compiler to generate code for the specific device (GPU), leveraging device's architecture advantages.

C. Similarities of CUDA and OpenCL

CUDA and OpenCL are parallel computing frameworks. CUDA is supported only on NVIDIA products, OpenCL has more general approach, it is cross-platform and supported on heterogeneous devices from different vendors. Since, OpenCL standard has been designed to reflect contemporary hardware there are a lot of similarities between CUDA and OpenCL frameworks.

OpenCL shares a set of core ideas with CUDA. These frameworks have similar platform models, memory models, execution models and programming models. Therefore, it is possible to transfer CUDA programs to OpenCL programs, and vice versa. Mapping between CUDA and OpenCL terminology, regarding memory and execution model, is presented in Table 1.

TABLE I.
CUDA VS. OPENCL TERMINOLOGY

CUDA	OpenCL
thread	*work-item*
block	*work-group*
global memory	*global memory*
constant memory	*constant memory*
shared memory	*local memory*
local memory	*private memory*

Mapping between CUDA and OpenCL syntax terminology is given in Table 2.

TABLE II.
CUDA VS. OPENCL SYNTAX

CUDA	OpenCL
__global__ (function)	__kernel
__device__ (function)	/
__constant__ (variable)	__constant
__device__ (variable)	__global

There are a lot of similarities in every aspect of these two programming frameworks, between CUDA and OpenCL. Almost every CUDA term can be mapped in OpenCL terminology. This fact led to creation of tools [8] for porting CUDA to OpenCL.

In this work, existing OpenCL code [6,7], was ported to CUDA, manually, following syntax and other mappings presented here.

Mapping between CUDA and OpenCL thread/work-item indexing is given in Table 3.

TABLE III.
CUDA VS. OPENCL THREAD/WORK-ITEM INDEXING

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
threadIdx + blockIdx*BlockDim	get_global_id()
gridDim*blockDim	get_global_size()

Mapping between CUDA and OpenCL API objects is shown in Table 4.

TABLE IV.
CUDA VS. OPENCL API OBJECTS

CUDA	OpenCL
CUdevice	cl_device_id
CUcontext	cl_context
CUmodule	cl_program
CUfunction	cl_kernel
CUdeviceptr	cl_mem
/	cl_command_queue

II. IMPLEMENTATION DETAILS

To CUDA/OpenCL programmer, the computing system consists of a host (often a CPU) and one or more devices (often GPU) that are massively parallel processors equipped with a large number of arithmetic execution units. Programs, that have been developed, use Java programming language for the host part of the computing system, and CUDA and OpenCL kernel's for programming of NVIDIA device that has been used.

A. Java and CUDA

In order to use Java as host programming language, we have used open source Java CUDA library (JCUDA ver. 0.5.5) [5]. This library gives a level of abstraction, between host and device calls/commands. Eclipse IDE has been used to create Java project and add JCUDA .jar files to project, after that .dll files have to be copied to location in the environment "path". Also, installation of CUDA toolkit (5.5) required an installation of MS Visual Studio (because it has bundled C compiler).

Kernel source code has to be compiled using NVCC compiler. As a result we have a file that we can load and execute using *Driver API*. There are two options how the kernel can be compiled: as a PTX file, as a CUBIN file. We have compiled our *kernel's* as PTX file, which is human readable (and not a case with CUBIN file).

B. Java and OpenCL

In order to use Java as host programming language, we have also used open source Java OpenCL library (JOCL ver. 0.1.3) [4]. We have used Eclipse IDE to create Java project, as with CUDA code. JOCL java archive files have been added to project path, and also JOCL.dll file has been put into environment "path". OpenCL is able to compile *kernel's* at runtime.

In both cases (CUDA and OpenCL) three kernel files have been create for: "*streaming*", "*collision*" and "*boundaries*". Execution of these *kernel's* have been called from host program written in Java (in cases of both frameworks). Performance results of these simulations are presented in next section.

III. PERFORMANCE RESULTS AND DISCUSSION

We have tested CUDA and OpenCL version of lid driven cavity (Figure 2) numerical simulation on NVIDIA GeForce GT 220. In Table 2 testing device (GPU) details have been listed. Latest CUDA drivers (320.57) and CUDA toolkit (5.5) have been used.

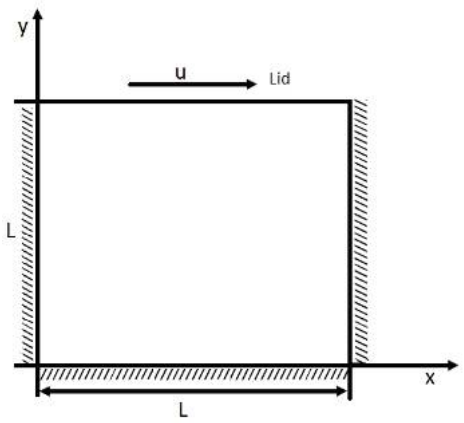

Figure 2. Lid driven cavity model

In Table 5. are listed device characteristics, which are in direct connection with performance of the simulation. Parameter number of compute units (*CL_DEVICE_MAX_COMPUTE_UNITS*) has proved to be a crucial for the execution speed in parallel algorithms.

TABLE V.
COMPUTE DEVICES CHARACTERISTICS

CL_DEVICE_NAME	**GeForce GT 220**
CL_DEVICE_GLOBAL_MEM_SIZE	**1 034 485 760**
CL_DEVICE_LOCAL_MEM_SIZE	**16384**
CL_DEVICE_MAX_WORK_ITEM_SIZES	**512** **512** **64**
CL_DEVICE_MAX_CLOCK_FREQUENCY	**1360**
CL_DEVICE_ADDRESS_BITS	**32**
CL_DEVICE_MAX_COMPUTE_UNITS	**6**

Platform and device information have been given in Table 6. These informations are 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 column are values obtained from the selected device for the demanded parameters.

The grid resolution used for this model ranges from 128x128 (16384 nodes) to 896x896 (802816 nodes). Steady-state of the simulation is achieved after approximately 20 000 time steps. Streamlines of the steady-state of the simulation, for Reynold's number 100 is shown on Figure 3.

TABLE VI.
TESTING DEVICE DETAILS

Vendor	**NVIDIA**
CL_DEVICE_NAME	**GeForce GT 220**
CL_PLATFORM_NAME	**NVIDIA CUDA**
CL_PLATFORM_VENDOR	**NVIDIA Corporation**
CL_DEVICE_VENDOR	**NVIDIA Corporation**
CL_DEVICE_TYPE	**GPU**
CL_DEVICE_OPENCL_C_VERSION	**OpenCL C 1.0**
CL_PLATFORM_VERSION	**OpenCL 1.1 CUDA 4.2.1**
CL_DEVICE_VERSION	**OpenCL 1.0 CUDA**
CL_DRIVER_VERSION	**320.57**

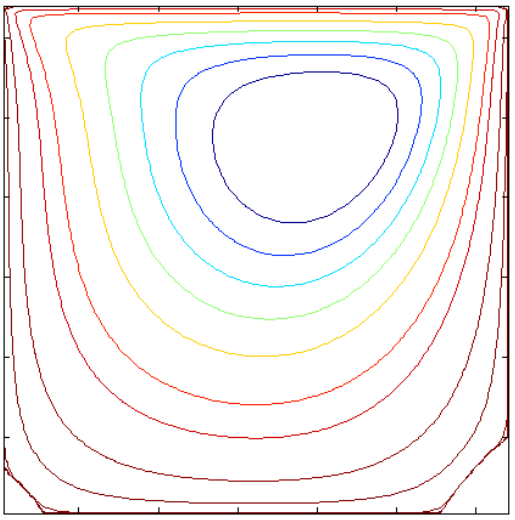

Figure 3. Streamlines Re=100

In Table 7. are displayed completion time (in milliseconds) for CUDA and OpenCL version of lid driven cavity numerical simulation.

TABLE VII.
RESULTS SIMULATION DURATION IN MILISECONDS

Mesh size	***Cuda***	**OpenCl**
128	43 934	22 564
256	77 512	59 031
368	140 527	118 648
512	221 731	199 347
768	443 835	478 044
896	558 605	566 954

Figure 4. represents graphical interpretation of the results given in previous table (Table 7.). From this figure we can draw a conclusion that for smaller mesh size OpenCL implementation is faster than CUDA implemetation of lid driven cavity numerical simulation.

OpenCl implementation performances dropped when mesh size was increased and became bigger then OpenCl local work size.

Simulation results show that there is no substantial difference in execution times between, almost identical, CUDA and OpenCL programs. Nevertheless, OpenCL program is portable, and can be executed on heterogeneous devices, from different vendors, without

any modification, which makes better choice for most developers.

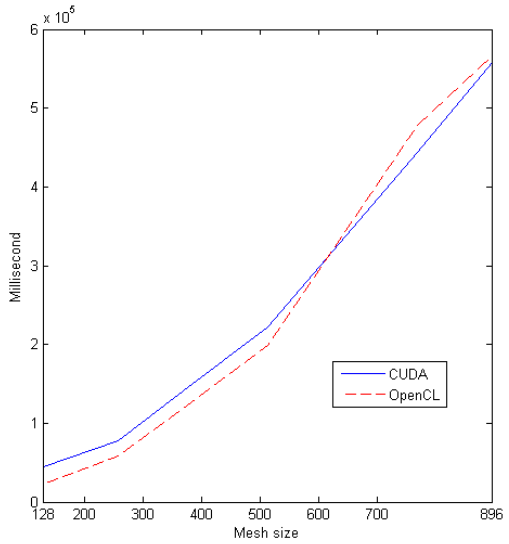

Figure 4. Execution time graph Miliseconds/Mash size

IV. CONCLUSION

In this paper we have compared performance results of two implementations, of the same benchmark problem, in two different GPU programming frameworks/interfaces, CUDA and OpenCL. We have used numerical simulation of a well-known benchmark problem (often used in CFD) lid driven cavity flow. Both implementations have used Java as "host" programming language from which both implementations made GPU calls.

Simulation has been carried out on NVIDIA GeForce GT 220 GPU. It has been shown that, although, CUDA is propritery framework developed for NVIDIA products, and OpenCL is portable between heterogeneous devices, simulation performance (time duration) is almost the same, and in some cases OpenCL program shows even better performance results then the similar CUDA program.

REFERENCES

[1] CUDA, http://www.nvidia.com/object/cuda_home_new.html, January 2014.

[2] OpenCL http://www.khronos.org/opencl/, January 2014.

[3] Khronos Group, http://www.khronos.org/, January 2014.

[4] Java OpenCL Library - JOCL, http://www.jocl.org/, January 2014.

[5] Java CUDA Library - JCUDA, http://www.jcuda.org/, January 2014.

[6] Tekić, P., Rađenović, J., Lukić, N., Popović, S., Lattice Boltzmann simulation of two-sided lid-driven flow in a staggered cavity, International Journal of Computational Fluid Dynamics, (ISSN 1061-8562), Vol. 24, Issue 9, pp. 383-390, 2010.

[7] Tekić, P., Rađenović, J., Racković, M., Implementation of the Lattice Boltzmann Method on Heterogeneous Hardware and Platforms using OpenCL, Advances in Electrical and Computer Engineering, (ISSN: 1582-7445), Vol. 12,No 1, pp. 51-56, 2012.

[8] Harvey, M.J., De Fabritiis, G., Swan: A tool for porting CUDA programs to OpenCL Computer Physics Communications, (ISSN 0010-4655), Volume 182, Issue 4, Pages 1093-1099, 2011.