Analysis of GPGPU Platforms Efficiency in General-Purpose Computations

P. Kartashev[1] **and V. Nazaruk**[1]
[1] Institute of Applied Computer Systems, Riga Technical University, Riga, Latvia

Abstract - *Nowadays a technique of using graphics processing units (GPUs) for general-purpose computing (or GPGPU) is becoming more and more widespread. The goal of this paper is to analyze efficiency of computing with use of the GPGPU technique, depending on several factors. In this paper, there are analyzed differences in performance and platform organization between widespread GPGPU computational platforms (both hardware and software). There are also described differences between CPU and GPU computations, as well as presented performance measurements for some GPGPU hardware architectures. This paper can help software developers choose more appropriate ways to implement specific fairly large computational tasks.*

Keywords: graphics processing units (GPUs), general-purpose computing on graphics processing units (GPGPU), OpenCL

1 Introduction

Many modern computers (approximately since 2006) have video cards that can be used not only for performing calculations connected with graphics, but also for arbitrary (even not related with graphics) calculations. Such technique of using graphic processing units for general-purpose calculations is called *general-purpose computing on graphics processing units (GPGPU)*.

Therefore, nowadays (in contrast to a period of several years before) most processing systems belong to one of the following two classes:

— central processing units (CPUs),
— graphics processing units (GPUs).

In order to use GPUs for computations, it is needed to write a program which uses a specific GPGPU programming model and architecture. Nowadays there exist several GPGPU platforms, which implement some different programming models and/or architectures; most notable of them include NVIDIA CUDA, OpenCL, Microsoft DirectCompute, ATI Stream.

For some kind of applications (usually for those which are multi-threaded and/or parallel), the use of general-purpose computations on modern GPUs can achieve speeds way beyond that on modern CPUs. Therefore, the use of graphics processing units for general-purpose computations is a topical sphere of research nowadays. The goal of this paper is to analyze efficiency of computing with use of GPGPU technique, depending on several factors, including target processing units, as well as GPGPU platforms themselves.

When speaking about the efficiency of GPGPU platforms, the thing that should be considered first is execution speed of programs which use the GPGPU technique. This mostly depends on specific processing units used for calculations, but also on a specific GPGPU platform architecture and programming model.

In this paper, there are analyzed and explained differences in performance and platform organization between GPGPU computational platforms (both hardware and software). Such GPGPU model comparison can help developers choose from these platforms to achieve best compatibility, speed, and portability for their GPGPU applications. Some guidelines for GPGPU developers, when they can use each of these platforms best, are formulated.

In the paper, there are also described differences between CPU and GPU computations. In our work, a comparison of GPU and CPU instructions is provided. There are presented performance measurements for GPGPU hardware architectures (including information about performance and time utilization of target processing units); some of the advantages and disadvantages of platforms are determined. Results concerning the performance measurements are based on practical experiments: by the authors, there was written and utilized an application (for the OpenCL programming model) for measuring the time of execution of different types of calculations. The methodology used is discussed further in this paper.

2 General-purpose computations on GPUs

A GPU is specialized for compute-intensive, highly parallel computation — exactly what graphics rendering does — and therefore designed in such a way that more transistors are devoted to data processing rather than data caching and flow control. A GPU is suited to problems that can be expressed as data-parallel computations — the same program is executed on many data elements in parallel — with high arithmetic intensity. Same program is executed for each data element, there is a lower requirement for sophisticated flow

control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D graphics, rendering large arrays of pixels and vertexes are processed in parallel are applied to parallel threads [1]. Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very fast. This approach of solving general purpose problems on GPUs is known as GPGPU. GPU has advantage over CPU by running data in parallel — benefit many tasks such as video/audio processing, large data sets processing, computational modelling (as industrial, weather, nature, particle simulation), ray-tracing, post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition. Many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology [1].

The main GPU advantage over CPU is its high throughput. Whilst CPU performance now increases only ~26% a year, GPU performance increases more than 100% a year.

GPU uses different architecture using many ALU units in the chip is the main difference from CPU, for example AMD PHENOM II X4 has 12 ALU, but GeForce GT240 GPU — 96 ALU (see Table 1).

GPU developers provide free GPU programming libraries (or SDKs), e. g. OpenCL, CUDA by Nvidia, Stream SDK by AMD.

CUDA (an acronym for "Compute Unified Device Architecture") is a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. CUDA is accessible to software developers through C for CUDA, CUDA Fortran Compiler and third party language wrappers, such as Jcuda, pyCUDA, etc. CUDA has been used to accelerate non-graphical applications. Programmers use C for CUDA (C with NVIDIA extensions and certain restrictions), compiled through NVCC compiler to code algorithms for execution on the GPU. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must spread across multiple disjoint memory spaces, unlike other C language runtime environments. Fermi GPUs now have (nearly) full support of C++ [2].

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL includes a language based on the C99 standard for writing kernels, plus APIs that are used to define and then control the GPGPU platforms. Programs written on OpenCL can access GPU of all supported GPU vendors for GPGPU computations. The OpenCL specification is under development by Khronos Consortium, which is open to everyone [3].

Microsoft's DirectCompute is a new GPU Computing API that runs under both Windows Vista and Windows 7. DirectCompute is supported on current DirectX 10 class GPUs, DirectX 11 GPUs. It allows developers to harness the massive parallel computing power of GPUs to create compelling computing applications in consumer and professional markets [4].

GPGPU platform comparison is described in further sections.

Table 1. Comparison of modern CPU and GPU used for the measurements in this paper

	AMD Phenom II X4	**NVIDIA GeForce GT240**
ALU in core / multiprocessor	3	8
Cores / Multiprocessors	4	12
Total ALU	12	96
Peak theoretical performance, GFLOPS	48 (3000 MHz)	385.9 (MADD+MUL) (3 instructions per cycle) (~1400 MHz)

3 Related works

In this section, there are discussed similar works to the research topic.

V. Volkov's work [5] shows that matrix manipulation with GPU can achieve speedup up to 2 times greater in double precision and 4–8 times for single precision than CPU. Comparison of performance is maintained. In [6], the authors provide a research study to achieve 10^3 speedup by using algorithm implementation with CUDA. Some works implement whole complex of algorithms on GPU: for example, the work [7] shows up to 20–100 times speedup, by implementing SQL-Lite SQL engine on CUDA architecture. The work [8] shows 20x speedup over CPU in AES cryptography. Research has been done by using NVIDIA CUDA. The work [9] proves that GPGPU computing problem is high PCI-E latency and low bandwidth, and sometimes optimizations required to achieve performance and there is big speedup in processing when using large data-set processing on GPU.

These works prove the efficiency of GPU based algorithms and describe useful uses of GPU. GPU has been used

mainly for scientific computations and large data processing. In this field, according to preceding works, GPU mostly outperforms CPU.

4 Comparison of GPGPU platforms

In this section, most widespread GPGPU programming models are described in context of comparison with each other.

We describe native programming language support and well as support for third party languages, for example Java, Python. We compare 3 main GPGPU platforms: OpenCL, CUDA, DirectCompute. We provide 3 criteria for the comparison:

1) Portability: what operating systems GPGPU computations could be made on?
2) Third party language support: is there support for GPGPU platform function call from other programming languages?
3) Possible execution on CPU (heterogeneous computing).

The weakness of DirectCompute is that it uses compute shader, which has specific restrictions: initialize a Direct3D device, create data buffers (resources) for shader, set shader state and launch it. Specific programming rules must be met.

As one can see, the DirectCompute API relies on DirectX 10 or 11 API to initialize GPU and make computations possible.

Despite DirectCompute benefits, that there is no need to special driver to make GPGPU computations using DirectCompute, and computations will run on every GPU that supports DirectX 10 or 11. DirectCompute is available for Windows Vista/7 only.

OpenCL and CUDA provide more flexibility and easy-to-write applications for GPU. We do not need compute shader to write and execute CUDA and OpenCL application. CUDA is available to only NVIDIA GPU's – application which was written for NVIDIA CUDA platform cannot be executed on ATI GPU's. OpenCL in comparison can be executed on various kinds or processing units, the only request is OpenCL driver from manufacturer of processing unit.

For the comparison summary, see Table 2.

Table 2. Summary of GPGPU programming model comparison

	OpenCL	CUDA	DirectCompute
Programming	C/C++ extensions	C/C++ extensions	C/C++, Shader Language
Portability	Windows, Linux, MacOS	Windows, Linux, MacOS	Windows Vista/7 with DirectX 10/11
API	OpenCL API	CUDA API	DirectX 11 API
Third party language support	yes (JOCL, PyOPENCL etc.)	yes (JCUDA, pyCUDA, Fortran PGI CUDA compiler etc.)	no
Heterogeneous computing possible?	yes	partial (only with program recompilation)	no (possible execution only on GPU)

5 Analysis of an impact of a GPGPU platform on computations

As it is stated before in this paper, use of GPGPU in an application can have an effect on the resulting characteristics of computations. The impact can be made, for example, by a target architecture for a GPGPU application. Such an impact is analyzed next; as well as further in this section, there is analyzed a possible impact of a platform on the performance of a GPGPU application.

5.1 Comparison of possible target hardware architectures for GPGPU: CPUs and GPUs

Platforms for general-purpose computing on graphics processing units (for example, OpenCL and CUDA) provide ways to execute an application written with a GPGPU technique also on computers where there are no GPGPU-compatible GPUs. In these cases, all instructions of the programs are executed on CPU — a GPGPU environment is imitated on CPU in a way that is transparent for an executing program.

This means that there exist two main target processing unit models for programs with GPGPU: when a program is executed both on a CPU and a GPGPU-enabled GPU, and when it is run only on a CPU. Therefore, it is important to compare which each other these two possible modes of operating for a GPGPU program.

For much software, the speed of their execution is of great importance. Bottlenecks for this speed usually are a processor (or processors) on which the software is executed, as well as memory and buses. However, when the software highly depends on calculations, or in the software there are many continuous uniform operations, the execution speed is mostly dependent on performance of the processing units.

In order to efficiently maximize the speed of execution of an application, a processing unit should be used to the extent possible.

All GPUs suitable for arbitrary calculations (i. e., with a support of GPGPU) are multi-core (for example, NVIDIA GeForce 580 GTX consists of 16 multiprocessors); and a large number of modern CPUs are also multi-core.

Modern GPUs, in contrast to CPUs, are composed of a large number of cores. Moreover, computational power of GPUs in average is comparable to (and mostly larger than) computational power of CPUs. This means that GPUs (as well as multi-core CPUs) provide a big possibility for speeding up execution of applications [10].

A majority of common algorithms are defined in a sequential way (i. e., the corresponding code of an algorithm is sequential). However, the fact that nowadays most of modern processors are multi-core assumes that for a specific sequential algorithm in order to execute efficiently, it should be parallelized — divided into several maximally independent (parallel) tasks. Thus, in order to take advantage of using multi-core processors, algorithms should be adapted for parallel execution [10].

Despite both CPUs and GPUs are multi-core, their architectures differ significantly. According to Flynn's taxonomy [11], multi-core CPUs have in general a *MIMD* (Multiple Instruction stream, Multiple Data stream) architecture, with each core usually having a support for a set of SIMD (Single Instruction, Multiple Data) instruction. Alternatively, all GPGPUs have a *SIMD* architecture [10].

The difference between the architectures of multi-core CPUs and GPGPUs substantiates differences between optimization processes for these two types of processing systems. However, all optimizations for a SIMD architecture are also applicable to a MIMD architecture — because a MIMD architecture can be considered as a more enriched SIMD architecture [10]. This means that when writing an application for a heterogeneous GPGPU programming model and targeting and correspondingly optimizing it for execution on a SIMD GPU, the optimizations will work and will have effect also when executing on a CPU.

There are differences between x86 CPU instructions and GPU instructions — GPU takes with one instruction also memory reference (address). This makes addressing more effective. Also GPU can deploy single instructions with many operands into SIMD array, which can consist of 8–512 (NVIDIA GPU) ALU. This makes developing parallel applications in a more effective way. In CUDA, OpenCL, DirectCompute, there is an emended native parallelism support. That make sense for example GPU executes parallel code 100 times faster than CPU, but CPU executes serial code 50 times faster than GPU. It is efficient to combine CPU and GPU to make possible heterogeneous computing with task divergence.

5.2 Analysis of performance of GPGPU applications

If one wants to use a processing unit to the maximal extent, before implementing an application it is good to know some guidelines, what actions which will perform the application are supposed to be fast, and which actions are supposed to be slow while running on a specific processing unit. In case of slow actions, at the application design stage, it is useful to avoid using slow operations to the extent possible. Therefore, it is useful to know, which operations on a specific application platform will perform faster, and which — slower.

In this section there are described practical results concerning speed of execution of primitive unary and binary operations (including basic arithmetical and bitwise Boolean operations) for three commonly used data types (*char*, *int*, and *float*) on the OpenCL GPGPU platform. OpenCL as a programming model was chosen mainly because of its support for ATI, NVIDIA GPUs and CPU, as well as multiple operating systems. OpenCL provides opportunity to run the same code on CPU and GPU.

For the measurement of speed, by the authors there was written a test application — an OpenCL program in the C++ programming language. To maximally smooth out the measurement errors, to measure small amounts of time with a high precision, each operation with the same input data was called 10 million times. This was implemented in a following way: an OpenCL kernel contained one operation (or a block of several similar operations), and the kernel was executed a specific number of times (for different data values) in a special loop (provided by an OpenCL programming model). The measurement of time intervals needed for the kernel to execute was implemented in the following way: the system time was measured just before and just after the execution of the kernel, and the difference of these values was considered as the execution time. The system time was measured using system calls, with the precision of several milliseconds.

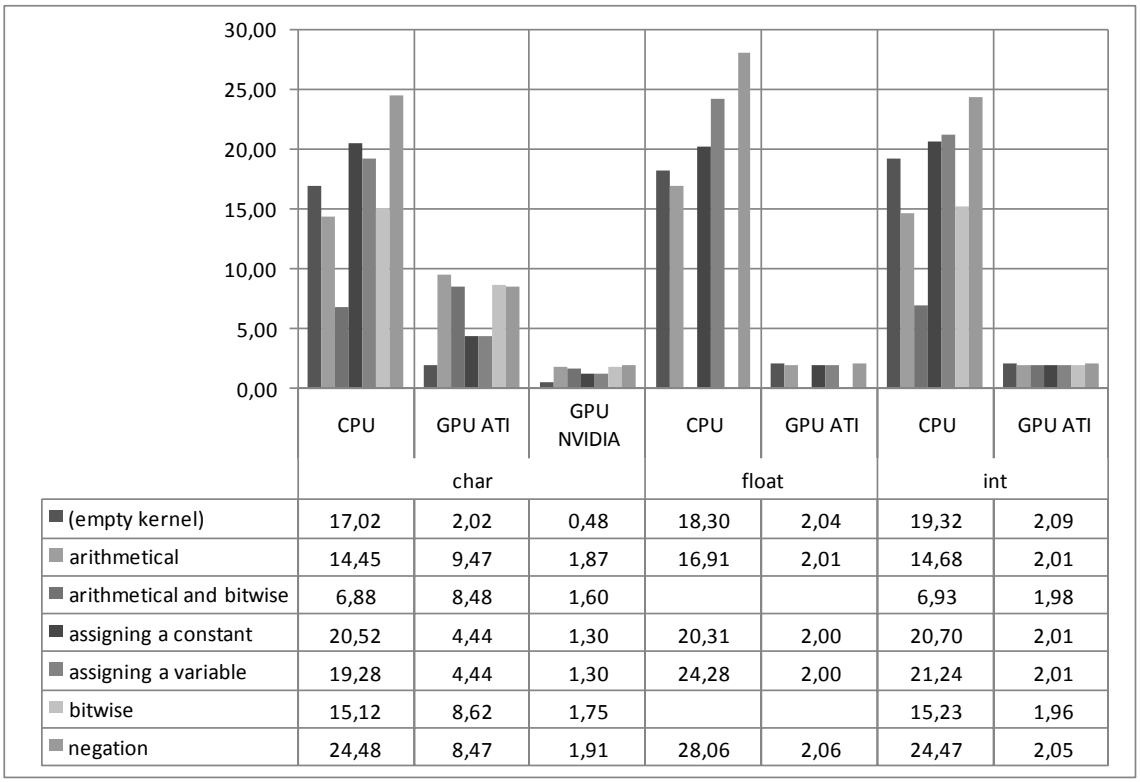
The test programs (32-bit) were executed on a computer with an AMD Phenom II X4 965 CPU (3.40 GHz in each of 4 cores), 4 GB RAM, and 64-bit Microsoft Windows 7 operating system, and the following video adapters:

— GPU ATI = ATI Radeon HD 5750,
— GPU NVIDIA = NVIDIA GeForce 240 GT.

With GPU NVIDIA due to technical problems there were measured only operations on the *char* data type.

It is needed to be stated that the obtained CPU performance is when forcing to run an OpenCL application on a CPU, not GPU. This means that in such a way obtained performance is not the same as the performance of a CPU when an application is implemented especially for running on CPU (i. e., without a use of a GPGPU).

The generalized results of the experiments in different views are provided in Figure 1–Figure 3.



	char			float		int	
	CPU	GPU ATI	GPU NVIDIA	CPU	GPU ATI	CPU	GPU ATI
(empty kernel)	17,02	2,02	0,48	18,30	2,04	19,32	2,09
arithmetical	14,45	9,47	1,87	16,91	2,01	14,68	2,01
arithmetical and bitwise	6,88	8,48	1,60			6,93	1,98
assigning a constant	20,52	4,44	1,30	20,31	2,00	20,70	2,01
assigning a variable	19,28	4,44	1,30	24,28	2,00	21,24	2,01
bitwise	15,12	8,62	1,75			15,23	1,96
negation	24,48	8,47	1,91	28,06	2,06	24,47	2,05

Figure 1. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit

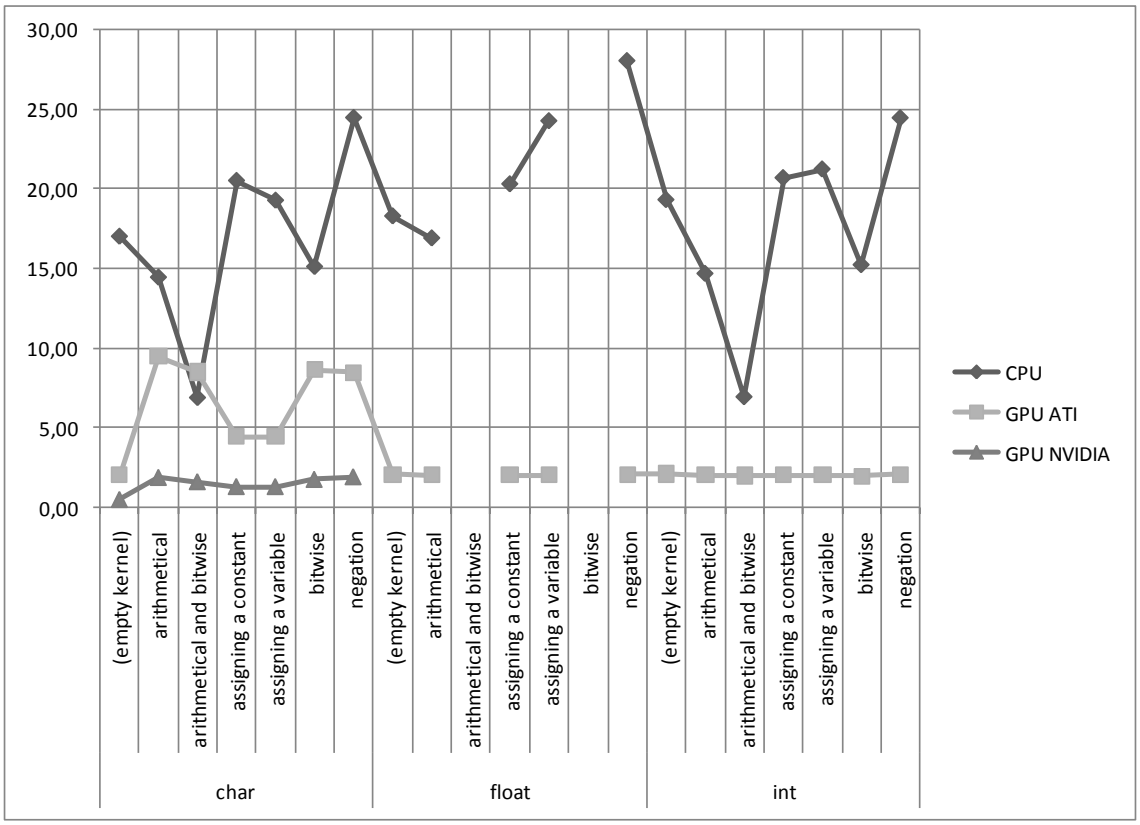


Figure 2. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit. It is easy to see that almost in all tested cases the fastest target processing unit is NVIDIA GPU, and the lowest — CPU

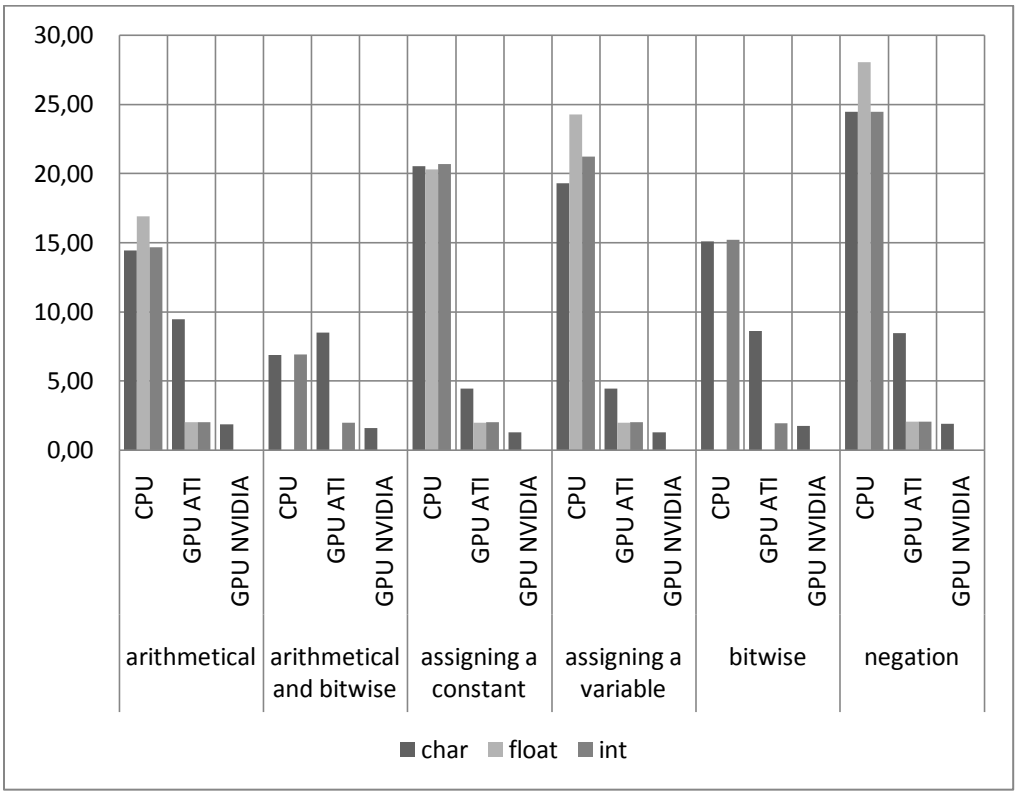


Figure 3. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit. One can see that on GPUs all operations with *char* data type (which takes integer values from -128 to 127) are significantly slower than operations with *int* and *float* data types. However, on GPUs operations with *float* data type are approximately as fast as with *int* data type; the same situation is on CPUs with all data types

From Figure 2 one can see that the fastest target processing unit is NVIDIA GPU, and the lowest — CPU. This is true almost for all operations. The performance of two different GPUs should not be compared directly; however, comparing the performance of the GPUs with a performance of the CPUs, one can see that the formers are much higher than the latter.

From Figure 3 one can see that on GPUs all operations with *char* data type (takes integer values from -128 to 127) are significantly slower than operations with *int* and *float* data types. This situation is slightly different from the usual situation with CPUs, where operations with *char* operands perform much faster than operations with *int* and *float* operands. Also, from the figure it is seen that, on GPUs operations with *float* data type are approximately as fast as with *int* data type; the same situation is on CPUs with all data types.

The differences between the obtained results showing GPU instruction performance, and generally known results on the performance of CPU instructions (including that floating-point operations on a CPU are performed much slower that integer operations) can be explained with the differences in the instruction sets and architectures of GPUs and CPUs. (For example, in [12], there is described instruction set architecture for ATI Evergreen Family GPUs).

6 Conclusions

When designing programs for GPUs which support general-purpose computing, in order to make programs be efficient, it is necessary to be aware of some specific features of GPUs. This includes the knowledge of the performance level of primitive mathematic operations — as there was shown in this paper, in calculations, it is better not to use variables of small size (i. e., of *char/byte* data type) but replace them with *integer*-type or floating point variables.

When intending an application for a GPGPU platform, it is needed to be known that in case of there is no GPGPU-enabled GPU on a destination computer, the performance will somewhat suffer. Speaking about GPGPU software platforms, it can be stated that OpenCL and CUDA provide more flexibility and easiness to write applications for GPU: there is no need for a compute shader to be used to execute CUDA and OpenCL app (unlike in DirectCompute). However, DirectCompute benefits from the point of view that there is no need to special driver to make GPGPU computations using DirectCompute, and computations will run on every GPU that supports DirectX 10, 11 or later. In addition to the above, in many cases the use of a specific GPGPU software platform (except OpenCL) can be limited either by GPU manufacturer (CUDA supports only NVIDIA GPUs, and Stream supports only ATI GPUs), or by a running operating system (DirectCompute cannot be run, for example, on Linux).

One of the directions of the further work is to improve the application used for the experiments in order to be able obtain wider measurement results (for example, not only for arithmetic or Boolean operations). Also, the experiments should be made on a larger set of different processing units.

GPGPU computations benefit in such tasks as image processing, physics simulations, large array processing, and many others tasks which deals with large data sets. A significant importance is information synchronization between threads that uses shared memory. As CPU cores can also use shared memory, it is possible for the further research to compare CPU and GPU data synchronization. Also the future research may include a new field of study — heterogeneous computing, which include both CPU and GPU computations. This is topical when a CPU and a GPU are combined on single die. Future research in the heterogeneous computing and APU (accelerated heterogeneous processing units) field may give results in understanding how to accelerate today's algorithms and programs in order for them to run faster on heterogeneous processors. In further studies, there could be also discussed the way how computations can be transferred between CPU and GPU, and how effectively a program written for GPU can be translated for running on CPU, and vice versa.

7 References

[1] "NVIDIA OpenCL Programing guide for the CUDA architecture, Version 3.2". [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Programming_Guide.pdf. [Accessed: Oct 2010].

[2] "CUDA - Wikipedia, the free encyclopedia". [Online]. Available: http://en.wikipedia.org/w/index.php?title=CUDA&oldid=389564200. [Accessed: Oct 2010].

[3] "OpenCL - Wikipedia, the free encyclopedia". [Online]. Available: http://en.wikipedia.org/w/index.php?title=OpenCL&oldid=389311710. [Accessed: Oct 2010].

[4] "DirectCompute - Wikipedia, the free encyclopedia". [Online]. Available: http://en.wikipedia.org/w/index.php?title=DirectCompute&oldid=389634399. [Accessed: Oct 2010].

[5] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", University of California at Berkley, SC08, November 2008. [Online]. Available: http://parlab.eecs.berkeley.edu/sites/all/parlab/files/Benchmarking%20GPUs%20to%20tune%20dense%20linear%20algebra.pdf. [Accessed: Oct 2010].

[6] A. C. Thompson, C. J. Fluke, D. G. Barnes, and B. R. Barsdell, "Teraflop per second gravitational lensing ray-shooting using graphics processing units", Centre for Astrophysics and Supercomputing, Swinburne University of Technology, May 2009. [Online]. Available: http://arxiv.org/pdf/0905.2453.pdf. [Accessed: Oct 2010].

[7] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA", Department of Computer Science University of Virginia, GPGPU-3, March 2010. [Online]. Available: http://www.cs.virginia.edu/~skadron/Papers/bakkum_sqlite_gpgpu10.pdf. [Accessed: Oct 2010].

[8] S. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", 2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24–27 November 2007, Dubai, United Arab Emirates: 2007. [Online]. Available: http://www.manavski.com/downloads/PID505889.pdf. [Accessed: Oct 2010].

[9] R. V. van Nieuwpoort, J. W. Romein, "Using Many-Core Hardware to Correlate Radio Astronomy Signals", Netherlands Institute for Radio Astronomy, 23rd ACM International Conference on Supercomputing. [Online]. Available: http://www.cs.vu.nl/~rob/papers/ics09-correlator.pdf. [Accessed: Oct 2010].

[10] V. Nazaruk and P. Rusakov, "Implementation of Cryptographic Algorithms in Software: An Analysis of the Effectiveness", Scientific Journal of Riga Technical University, Vol. 43, pp. 97–105, 2010.

[11] M. Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Transactions on Computers, Vol. C-21, Issue 9, pp. 948–960, 1972.

[12] "Evergreen Family Instruction Set Architecture. Instructions and Microcode. Reference Guide", Advanced Micro Devices, Inc., September 2010. [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf. [Accessed: Oct 2010].