Software for heterogeneous computer systems and structures of data processing systems with increased performance

A.A. Kolpakov[1], Ju.A. Kropotov[1]

[1]*Murom institute (branch) VlSU, Orlovskaya st., 23, 602264, Murom, Vladimir region, Russia*

Abstract

The issue of creating high-performance computing systems based on heterogeneous computer systems is topical, as the volumes of processed information, calculations and studies with large data sets are constantly increasing. The aim is to develop software design techniques heterogeneous computer data processing system. As a result, a technique has been developed for combining shaders to improve the performance of heterogeneous computations. The presented solution is supposed to be implemented as a software module for a computer system using CUDA technology.

Keywords: parallel computing; heterogeneous computing systems; graphics processors; CUDA; OpenCL

1. Introduction

The GPGPU program can be conditionally represented using the following sets:

- set of shaders that perform calculations;
- set of variables that control the computations;
- the set of data over which the calculations are performed and in which their results are recorded;
- a set of instructions that run a particular shader, provide him with input for certain data and output the result to a certain texture.

1.1 GPU programming based on vertex and pixel programs

The elementary primitive for visualization, with which the graphics processor works, is a triangle. With each vertex of a triangle, it is possible to associate a limited set of arbitrary data, for example, its color, normal, and other user data, Up to 8 textures can be associated with the primitive itself – one-, two-, and three-dimensional images. The structural scheme of data processing based on vertex and pixel programs is shown in fig. 1.

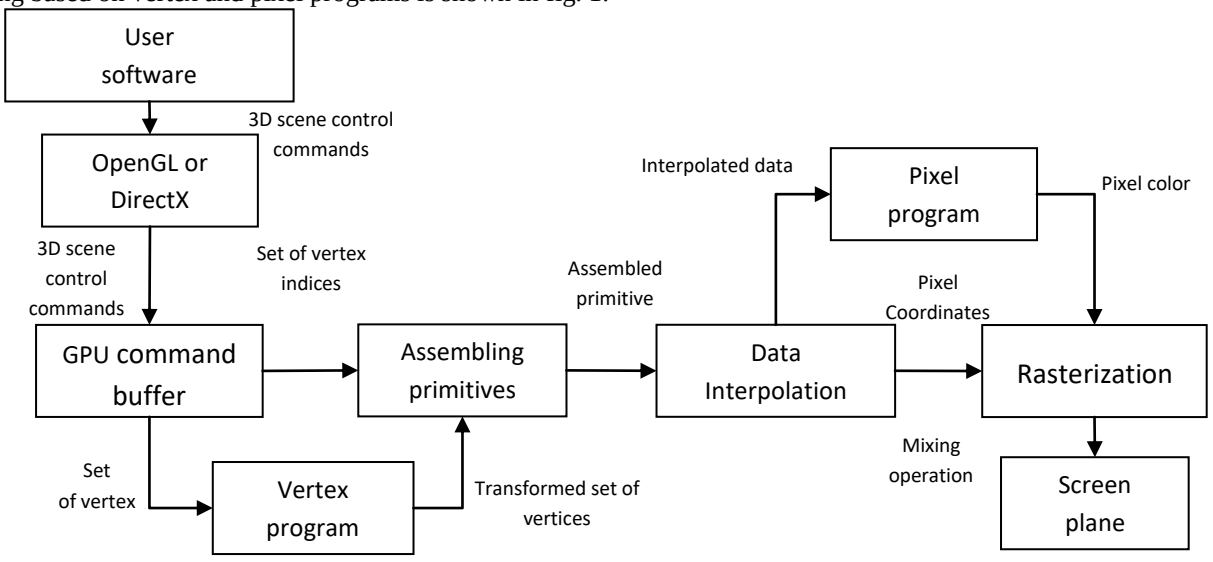


Fig. 1. The structural scheme of data processing based on vertex and pixel programs.

As shown in fig. 1, the user application sends requests for visualization of a 3D-scene to the OpenGL or DirectX low-level programming libraries, which are parts of the operating system. Then this data is transformed with the graphics card driver into direct commands of the graphics processor [1,2,3].

1.2 GPU programming based on CUDA library

Despite the fact that programming vertex and pixel programs has proven effective for modeling various physical processes, the use of this method is not convenient for the programmer. The programmer needs to have a sufficiently high qualification to perform computational tasks on the graphics processor, i.e. to use the principles of the GPU in detail, because a small inaccuracy

in the control code of the graphics processor can lead to a significant distortion of the result of the calculations. The structural scheme of the software on the basis of CUDA library is shown in fig. 2.

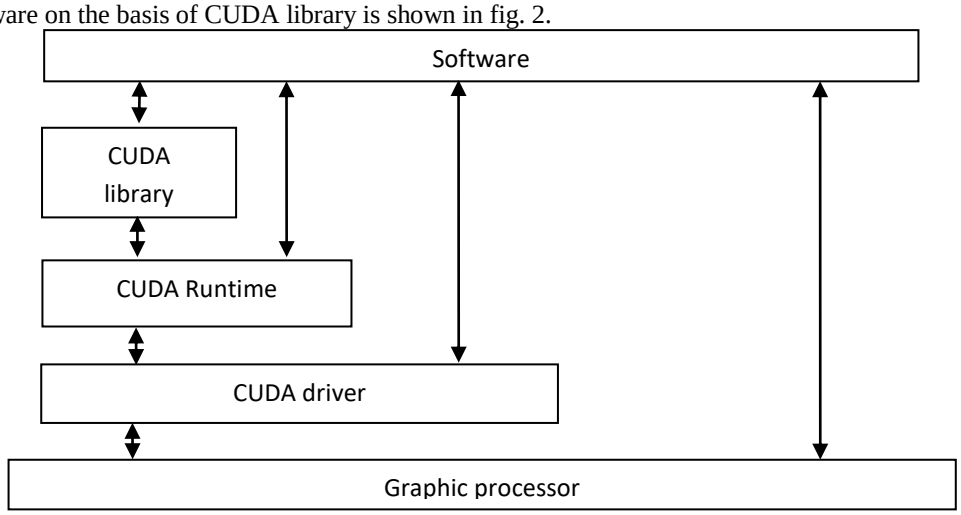


Fig.2. The structural scheme of the software on the basis of CUDA library.

The executable unit of the CUDA program is warp. The size of warp is 32 threads. This is due to the fact that latency is 4 cycles when executing one instruction on the multiprocessor. Only with respect to warp we can talk about the parallel execution of flows, no other assumptions can be made. However, this does not mean that warps are executed sequentially on the multiprocessor. The execution of warps can be parallel, for example, in the event that one warp expects data from global memory, other warps can be executed at this time.

Interaction between threads can be carried out only within the block. Data is exchanged via a shared memory which is common to all streams in the block. The execution of threads can synchronize by calling special synchronization functions.

1.3 GPU programming based on OpenCL library

The development of the GPU and its use in tasks unrelated to computer graphics has resulted in the development of a single standard for describing computations on highly parallel systems – OpenCL (Open Computational Library). The generated OpenCL library appeared on the basis of the previously developed Nvidia CUDA technology, which describes the interface of application interaction with the computing resources of the graphics processor. Unlike CUDA technology, OpenCL technology describes a computation model without connection to a specific type of device on which these calculations will be executed. Due to the fact that OpenCL is designed exactly as a standard for computations on highly parallel systems, many specific features of CUDA technology have been excluded from the standard. In general, CUDA technology has more possibilities, in comparison with OpenCL for describing parallel computing, if the Nvidia graphics processor is the computing device.

OpenCL allows to describe the calculations, abstracting from a particular device, on which these calculations will be implemented. In general, OpenCL algorithms can be executed on several CPU cores, on a graphics processor or on IBM Cell / B.E processors. OpenCL implementation uses extensions of the C language to describe the algorithm [3,4].

The OpenCL library is a promising library for use in various scientific research. The advantage of the OpenCL library is support from high-performance clusters. Any application that uses OpenCL can be run without modifications on a cluster that contains, among other things, graphics processors. This application will be available to all existing computing resources in the system. The structural scheme of the software on the basis of the OpenCL library is shown in fig. 3.

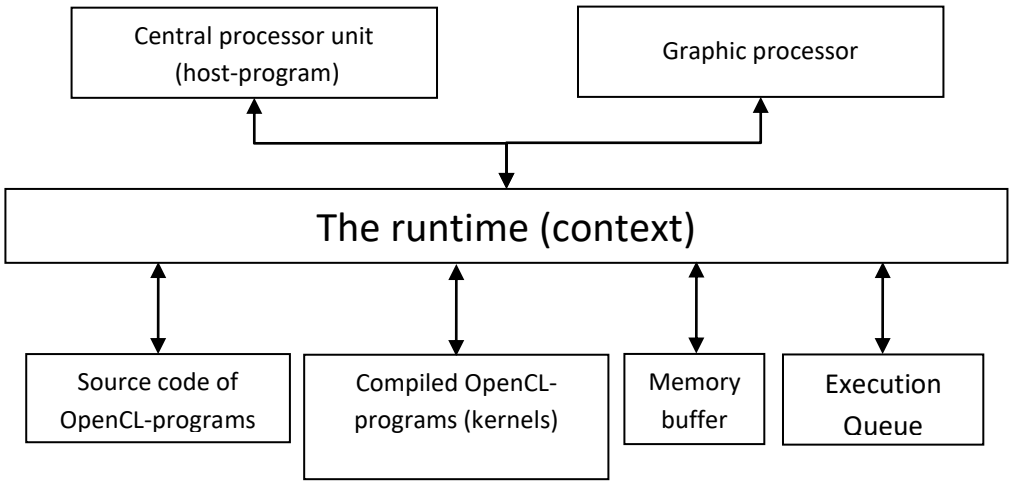


Fig. 3. The structural scheme of the software on the basis of the OpenCL library.

The central element of the OpenCL platform model is the concept of a host, the primary device that manages OpenCL calculations and performs all interactions with the user. The host is always represented in a single instance, while the OpenCL-devices on which the OpenCL-instructions are executed can be represented in the plural. OpenCL-device can be CPU, GPU, DSP or any other processor in the system, supported by the OpenCL-drivers installed in the system.

2. Software of a heterogeneous computer data processing system

The block diagram of the developed software of a heterogeneous computer data processing system is shown in fig. 4.

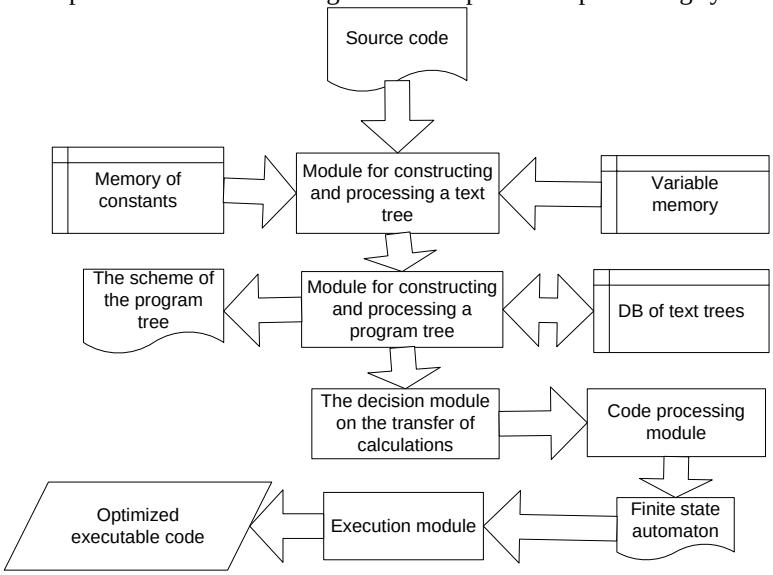


Fig. 4. The block diagram of the developed software of a heterogeneous computer data processing system.

According to fig. 4 the input source for the software is the source code of the program being processed. It enters the module for building and processing a text tree, in which the initial processing of the source code occurs and the construction of a text tree on its basis. Memory constants and variable memory are used to work with a text tree.

The received text tree is transferred for processing to the module for constructing and processing the program tree. The module for building a software tree uses a database of previously processed text trees, which allows to significantly accelerate the construction of a program tree [5,6].

For the processing of the program it is necessary to build a tree that represents a text description in a convenient format. There can be several types of nodes in the program tree:

- root node – contains all the data that must be computed during the execution of the program as descendants;
- node with data – represents an array of data that is required to transmit the input ancestor or into which to write the result descendant;
- node with variable – represents a variable that must be passed to the input of the ancestor shader or that describes the condition of the conditional parent operator;
- node with no operations – transfers the data of the descendant to the ancestor without changing them;
- node with shader – performs a shader with input data received from the children and passes the result to the ancestor;
- node with arithmetic operation – performs arithmetic transformation of input data received from descendants and passes the result to the ancestor;
- node with branching start – describes a conditional statement; the first child is a variable that describes which branch to choose, followed by the various branches of execution;
- node with branching completed – describes the completion of the conditional statement, all its branches are reduced to this node.

Trees are alternately built for each instruction. The parser parses the string representation, defines the output array with the data, optionally expands the abstractions and defines the set of functions that must be performed to obtain the result.

In each subsequent tree, the node with the data that was recorded in one of their previous instructions is replaced by the tree of the last instruction. The program tree is obtained after processing the last instruction. It should be noted that the software tree may not be a tree by definition - some nodes may have more than one ancestor, but most often it represents a tree or a structure close to it.

The software tree is transferred to the module of decision about the transfer of calculations, where the original algorithm is divided into stages and a decision is made to transfer the calculations to the GPU for each stage with further transfer to the code processing module.

3. Development of the code processing module

The code processing module makes the necessary changes to the source code program to produce a finite automaton of the program transmitted in the execution unit, generating executable code treated.

The level of the target executor for the family of accelerators takes on the input a graph of functional operations and on the output receives a program for the GPU [7,8]. This program has the form of a performance script, in which operations are available for starting the shader on the GPU with a certain set of parameters, allocating and freeing memory, transferring data from RAM to video RAM and back. The basic stages of broadcasting the program at the level of the target performer:

1. Splitting the original graph into subgraphs corresponding to different passes. Subsequently each subgraph is broadcast separately.

2. Select the display for the data and for the code. In this case, for example, one array of logical data can be mapped to several physical GPU buffers, and the restore operation in the source code is performed in a loop, with each iteration processing a block of 4 elements.

3. Code generation for the GPU. After the mappings are selected, they are generated by the code on the GPU. In particular, the indexes for the GPU buffers are calculated and the repetitive reads are eliminated.

4. Conversion of the generated code. At this stage, the arithmetic expressions are merged into vector commands and the constants are convoluted.

Since the individual expressions calculated on the GPU are relatively small, and the passage requires a large computing power of the shader, the first stage is relatively simple. Most often, the result of his work is the only subgraph that coincides with the original graph. The main criterion for partitioning into passages is the reuse of data. Splitting is only possible if you can not avoid multiple calculations of the same data without it.

The most difficult from the point of view of implementation are the stages 2 and 3. Step 4 is relatively simple and is performed using arithmetic transformations. The possibility of merging arithmetic expressions into vector operations is provided by an efficient choice of mappings in stages 2 and 3.

The functional graph C$ defines the following types of vertices [9]:

Sheet tops:

1. Constants.

2. Related variables. In fact, are similar to the cycle. They run through a certain range and can be used to calculate the index expressions.

3. Functions. They can be member functions (including standard operations) or arrays. If the function is found expression, it is applied to its arguments, so in the end it will not be presenting.

Internal vertices:

5. Applying a function to arguments (@). This can be an application of a normal function or a reference to an array element.

6. Reduction operation (R). It has 2 arguments, the first of which specifies a reducing function, and the second one is reducible. Reduction is applied to all dimensions of an array that does not contain associated variables, as well as related variables, which allows partial reduction.

Associated variables "spread" from the bottom up the graph. They can end their distribution only at the vertices of reduction or at the root apex. We also assume that an acyclic graph is arranged in such a way that all paths of propagation of a bound variable terminate on the same vertex.

Below is an example of combining two shaders, in which you can see the non-optimal parts of the code. The block diagram of the first shader is shown in fig. 5.

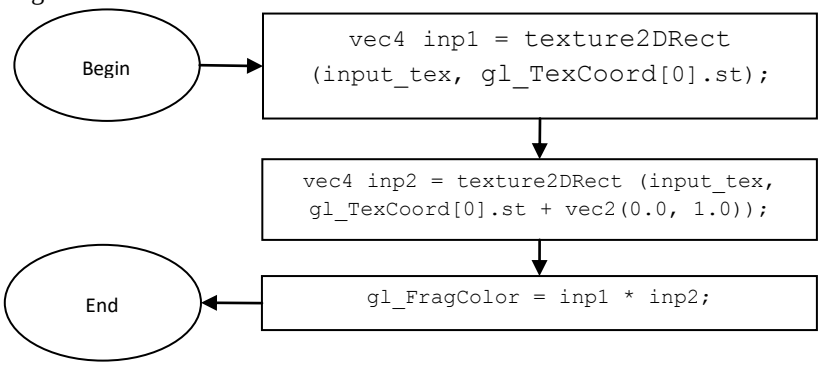


Fig. 5. Block diagram of the first shader.

A block diagram of the second shader is shown in fig. 6.

The block diagram of the combined shader, obtained after merging the sequence of these shaders, is shown in fig. 7.

To perform the shader more efficiently, it is necessary to perform its correction. A block diagram of the final adjusted shader is shown in fig. 8.

After carrying out all the above-described changes, the shader code will be compiled and will be ready for use in calculations.

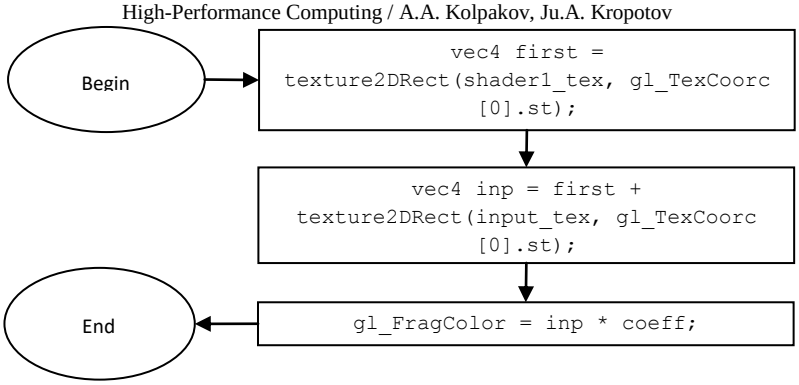


Fig. 6. Block diagram of the second shader.

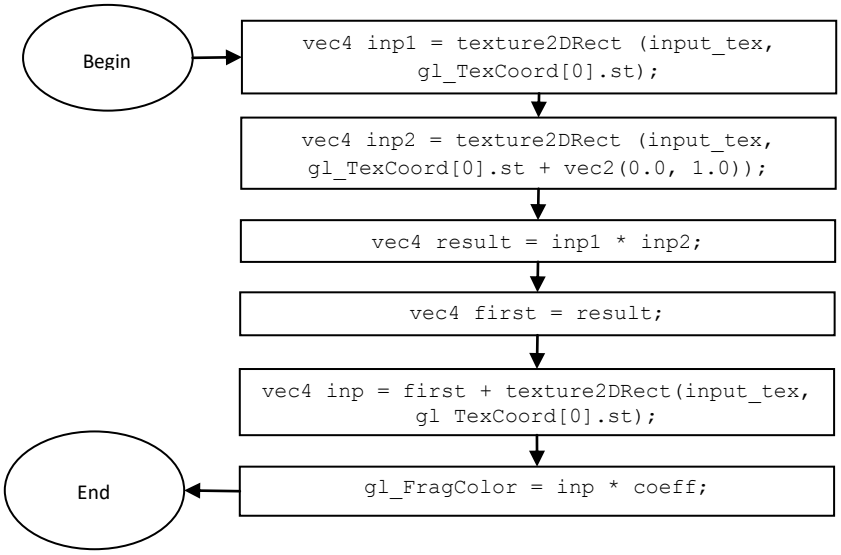


Fig. 7. Block diagram of the combined shader.

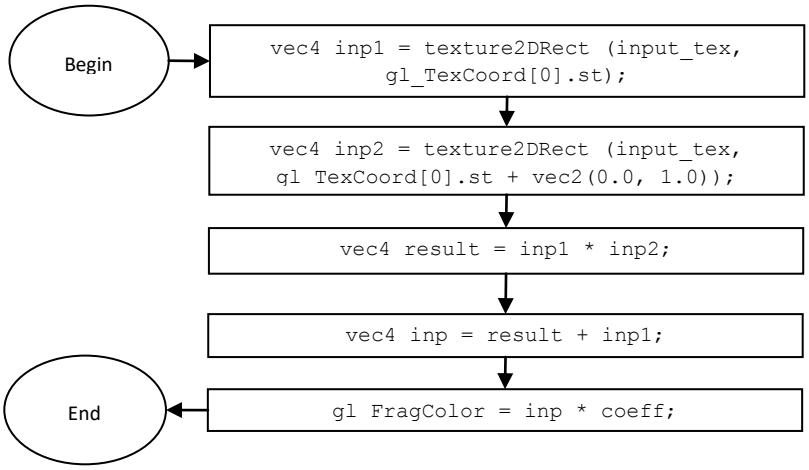


Fig. 8. Block diagram of the resulting shader.

4. Conclusion

The program provides the schemas of the processed program tree and the generated automaton and the received shaders as output. The scheme of the processed program tree is shown in Fig. 9.

As can be seen from fig. 9, the software tree consists of nodes of various types, each of which describes the behavior of the program: the root node; node with data; node with variable; node with no operations; node with a shader; node with arithmetic operation; node with the beginning of branching; node with the completion of branching.

The code for the graphics processor is generated in accordance with the selected display. Each key vertex is associated with a set of output variables, for the calculation of which it responds. The root node, in addition to their calculation, is responsible for writing them to the output buffers. For each sub-graph of operations, a code template is generated. Based on this template, a code is generated for each set of values of the block measurement instances [10].

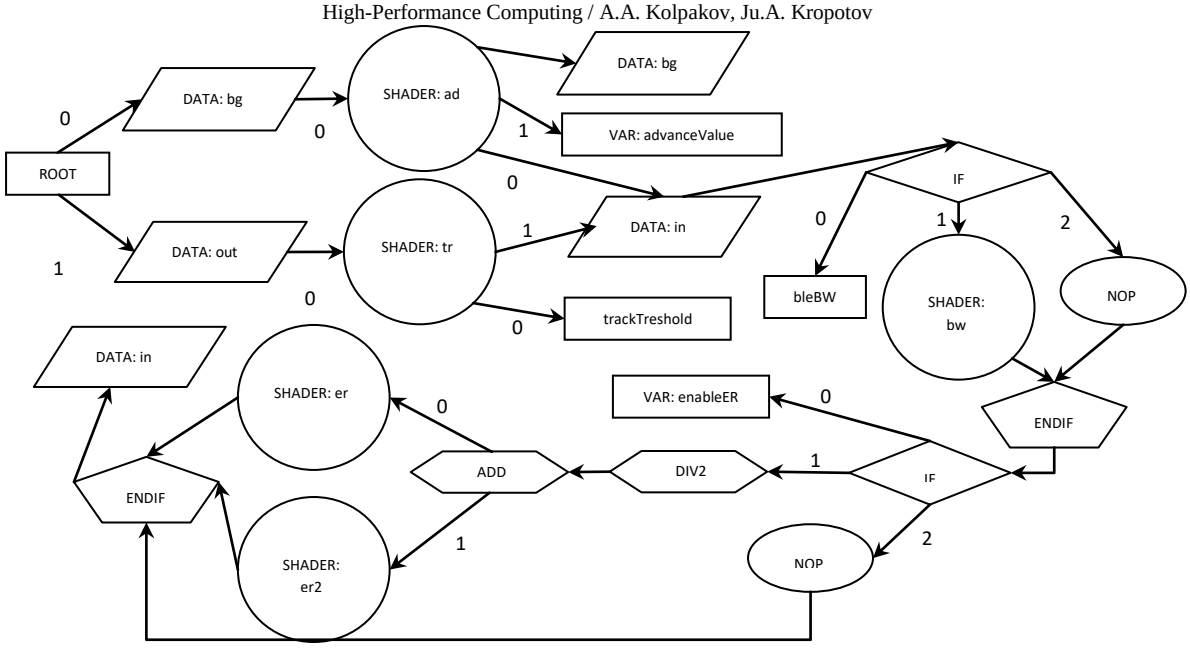


Fig. 9. Processed software tree.

The most difficult part of the code generation phase is the generation of access commands in RAM. By mapping, the number of buffer numbers that can participate in this data reading operation is determined. Further, a set of two-dimensional indexes for reading from this buffer is determined for different values of block indexes for each of the buffer numbers. There is the vertex closest to the root, in which the index can be calculated, for each index, based on a set of variables on which it really depends. Improves the calculation of indices, which are linear by iteration of the cycle. In each key vertex, a set of indices is marked, which it needs to compute, in order to avoid the recalculation of the indices. The data reading operation is performed in the same key node to which the index belongs. This allows you to avoid re-reading the same data from the RAM, even if they are made by different reading vertices.

To work with the indices used automatic conversion tools of integral and simplifying expressions. They support the conversion of expressions containing arithmetic operations, minimum, maximum, and comparison operations.

The program allows flexible manipulation of options.

In the course of investigating the effectiveness of the developed information processing algorithms, the problem of finding zero bit vectors was solved, which is solved using genetic algorithms. In solving this problem, the main working time is occupied by parallel computations of the values of the fitness functions of different individuals, the operations of crossing and mutation. The algorithm has properties that are characteristic of many genetic algorithms:

1. Representation of an individual in the form of a bit string.
2. A small number of logical operations in calculating the function of fitness, performing mutation and crossing.
3. Consecutive memory access.

These properties allow to effectively use the calculations on the graphics processor.

The mutation operation is standard for such individuals - changing the value of one random bit. A single-point crossover is used as a cross operation.

The function of fitness of an individual is the number of units in it. Accordingly, it is necessary to deduce an ideal individual with a zero fitness function. There is an algorithm that allows you to calculate the number of units in a 32-bit number using only arithmetic operations.

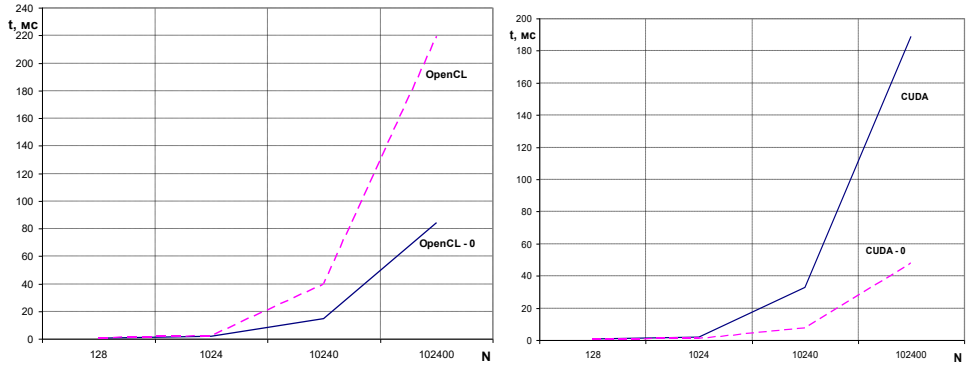
To test the efficiency of the algorithm for improving data processing efficiency, we used a test computer system of the following configuration: CPU Intel Core 2 Quad Q9400 (2.66GHz), 8GB RAM, graphics card Nvidia GeForce GTX560 2Gb with 336 streams, Windows 7 x64 operating system, MS compiler Visual Studio 2008 in release mode.

The average time spent on receiving a new generation for different sizes of the problem and the number of individuals in the generation was investigated. For this, several iterations of obtaining the next generation (about 100-1000 starts) were started and the total time spent on the whole algorithm was divided by the number of generations obtained.

Performance study In the first challenge test varied number of 32-bit integers in the array (M) and the number of parallel streams (N).

We investigated the average time t spent on obtaining a new generation for a different number of 32-bit integers in an array and the number of parallel threads. The research was conducted using OpenCL and NVIDIA CUDA technologies [4]. The results of the study of the average time spent on obtaining a new generation for the parameter M = 10 are shown in Fig. 10.

In fig. 10 graphics OpenCL and CUDA show the execution time of the base algorithm, OpenCL-O and CUDA-O - using the developed algorithm for improving data processing performance. As can be seen from the graphs, with the value of the parameter M = 10, the application of the developed optimization algorithm gives an increase in performance: in the case of OpenCL, the processing time for 128 threads is reduced from 1.08 ms to 0.75 ms and from 219 ms to 84.2 ms for 102400 Flows. In the case of NVIDIA CUDA, the processing time is reduced from 0.85 ms to 0.74 ms for 128 threads and from 189 ms to 48.4 ms for 102400 streams [4].



a b

Fig. 10. The average time spent by a parallel system to receive a new generation, whith M =10, a: OpenCL – basic algorithm, OpenCL-O – developed algorithm, b: CUDA – basic algorithm, CUDA-O – developed algorithm.

References

[1] Bahvalov NS, Voevodin VV. Modern Problems of Computational Mathematics and Mathematical Modelling. Computational Mathematics Book. Vol. 1. Moscow, Science, 2005; 342 p. (in Russian)

[2] Boreskov AV. Shaders development and debugging. Sankt-Peterburg: BHV-Peterburg, 2006; 488 p. (in Russian)

[3] Kolpakov AA. Theoretical evaluation of growth performance computing systems from the use of multiple computing devices. V mire nauchnykh otkrytii 2012; 1: 206–209. (in Russian)

[4] Kolpakov AA. Optimizing the use of genetic algorithms for computing graphics processors for the problem of zero bit vector. Informatsionnye sistemy i tekhnologii 2013; 2(76): 22–28. (in Russian)

[5] Barkalov KA, Gergel VP. Parallel global optimization on GPU. Journal of Global Optimization 2016; 66(1): 3–20.

[6] Strongin RG, Gergel VP. Parallel computing for globally optimal decision making on cluster systems. Future Generation Computer Systems 2005; 21: 673–678.

[7] Galimov MR, Biryalcev EV. Some technological aspects of GPGPU applications in applied program systems. Vychislitelnye metody i programmirovanie 2010; 11: 77–93. (in Russian)

[8] Kropotov JuA, Belov AA, Proskuryakov AJu, Kolpakov AA. Methods of Designing Telecommunication Information and Control Audio Exchange Systems in Difficult Noise Conditions. Sistemy upravleniia, sviazi i bezopasnosti 2015; 2: 165–183. (in Russian)

[9] Boreskov AV. OpenGL extension. Sankt-Peterburg: BHV-Peterburg, 2005; 672 p. (in Russian)

[10] Ermolaev VA, Kropotov JuA. Algorithms for processing acoustic signals in telecommunication systems by local parametric methods of analysis. Proceedings International Siberian Conference on Control and Communications (SIBCON), 2015. URL: http://ieeexplore.ieee.org/document/7147109/.