APPLICATION OF THE OPPORTUNITIES OF TOOL SYSTEM "CUDA" FOR GRAPHIC PROCESSORS PROGRAMMING IN SCIENTIFIC AND TECHNICAL CALCULATION TASKS

V.A. Dudnik, *V.I. Kudryavtsev, T.M. Sereda, S.A. Us, M.V. Shestakov

National Science Center "Kharkov Institute of Physics and Technology", 61108, Kharkov, Ukraine

(Received May 27, 2009)

The opportunities of technology CUDA (Compute Unified Device Architecture - the unified hardware-software decision for parallel calculations on GPU)of the company NVIDIA were described. The basic differences of the programming language "C" for GPU from "usual" language "C" were selected. The examples of CUDA usage for acceleration of development of applications and realization of algorithms of scientific and technical calculations were given which are carried out by the means of graphic processors (GPGPU) of accelerators GeForce of the eighth generation. The recommendations on optimization of the programs using GPU were resulted.

PACS: 89.80.+h, 89.70.+c, 01.10.Hx

1. INTRODUCTION

Programming of the decisions of intensive mathematical tasks, which is given an opportunity of usage of existing graphic processors for acceleration of their decision is not only new and perspective, but also roughly developing tendency of development of tool means for the software. The research of opportunities of such one technical decision - technologies *CUDA (Compute Unified Device Architecture - the unified hardware-software decision for parallel calculations on GPU)*, offered by the company NVIDIA, is very actual in this connection. This technology represents C-like programming language with the compiler and libraries for calculations on GPU for development of appendix and gives the to programmer greater control over hardware opportunities GPU. [1],[2] It is important, that support of NVIDIA CUDA is at chips G8x, G9x and GT2xx, series 8 used in the video adapters GeForce, 9 and 200 which are widely distributed. Besides it is necessary to pay attention, that these software of the development and their description are given completely free-of-charge (SDK for all basic platforms is freely downloaded with developer.nvidia.com). It does their usage especially attractive for scientific researches. Once more key moment of architecture CUDA is easy scalability. Once on as written code will be started on all devices supporting CUDA[9]. For the development and debugging of a code for start on GPU it is possible to use usual video adapters. When the product is ready it is possible to start it on more powerful GPU. It is necessary to note that parallel processing on GPU some differs from the work with CPU. It is said about parallelism of tasks within the development of applications for traditional CPU i.e. one program module is executed on the first processor (or a nucleus), another is executed on the second, etc. Parallel processing by the means use CUDA assumes the parallelism of the data, i.e. presence of a plenty (much more, than physical processors GPU) elements or groups of the data which admitting the independent parallel processing. The purpose of given work was the research of opportunities of the technology CUDA for applications' development and algorithms'realizations of scientific and technical calculations in NSC the KIPT.

2. THE GRAPHIC PROCESSOR AS THE SIMD SET OF MULTIPROCESSORS

Development of the functionalities of graphic processors (particularly blocks for calculations of pixel shaders) has made possible the usage of video adapters for scientific and technical calculations - as a powerful SIMD (Single Instruction Multiple Data) processors. Occurrence of the technologies of not-graphic general purpose calculations **GPGPU (General-Purpose computation on GPU)** promoted it. With the help of these technologies there was possible to use hundred mathematical **executive** blocks of modern video chips GPU as general purpose processors for the significant acceleration computational intensive applications. For understanding of features of programming GPGPU it is necessary to take into consideration the features of structure GPU (Fig.1) and its work in the basic

*Corresponding author E-mail address: vladimir_1953@mail.ru mode (it is real this graphic device, and its basic purpose is formation of the image). As you can see on Fig.1, the part of the graphic processor NVIDIA[4], used for GPGPU - the nucleus of shaders - consists of several clusters of textural processors **(Texture Processor Cluster, TPC)**.

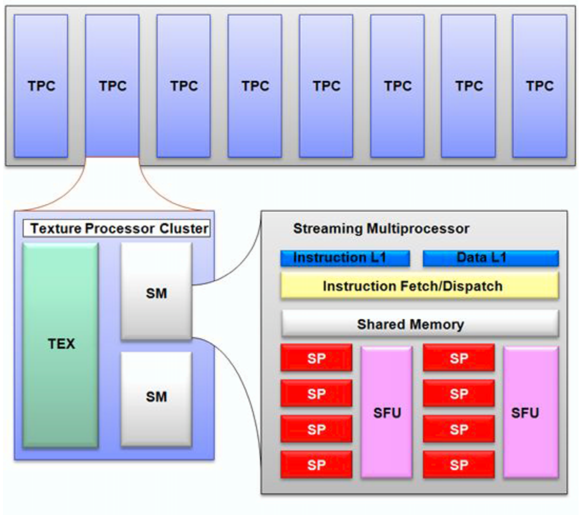

Fig.1. *Structure of a nucleus of shaders*

Quantity of TPC in the device depends on the model GPU (and its price, accordingly). The video chip 8800 GTX, for example, contains eight clusters, six 8800 GTS, etc. Each cluster consists of the textural block and two or three streaming multiprocessors (streaming multiprocessor, SM). The streaming multiprocessors include 16 kB shared memory (Shared Memory, it is not cache-memory: the programmer can use it at own discretion), eight computers (Streaming Processors, SP)) and two super functional devices SFU (Super Function Unit) where instructions are carried out by the principle SIMD, i.e. one instruction is applied to all elements of the data. NVIDIA shows such way of performance SIMT (**single instruction multiple threads** - one instruction, it is a lot of streams). Shared memory gives the opportunity of information interchange between streams in one block. In the graphic mode the blocks of pixel shaders work as follows (see Fig.2): the block of geometry generates triangles, then the block of rasterization generates **quads** - squares of pixels 2x2 where each pixel is set by a vector with four values from a floating point of unary accuracy (R, G, B, A)or (X, Y, Z, W) - most often used format in 3D-calculations. The quads then act in streaming processors **(SM)**(see Fig.3) (which work in 16-channel mode SIMD, i.e. the identical instruction is applied to all 16 numbers from a floating point. When 8 quads (32 pixels,**"warp"** on terminology CUDA) are collected in the buffer, they are carried out by the multiprocessor in mode SIMD, and then entrance data from structures for each pixel are read out, are developed and are entered the name in the target buffer.

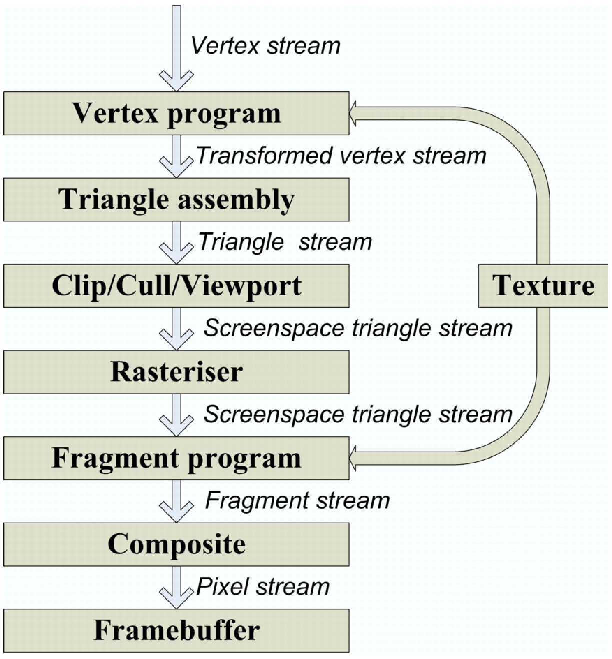

Fig.2. *Circuit of work of pixel shaders*

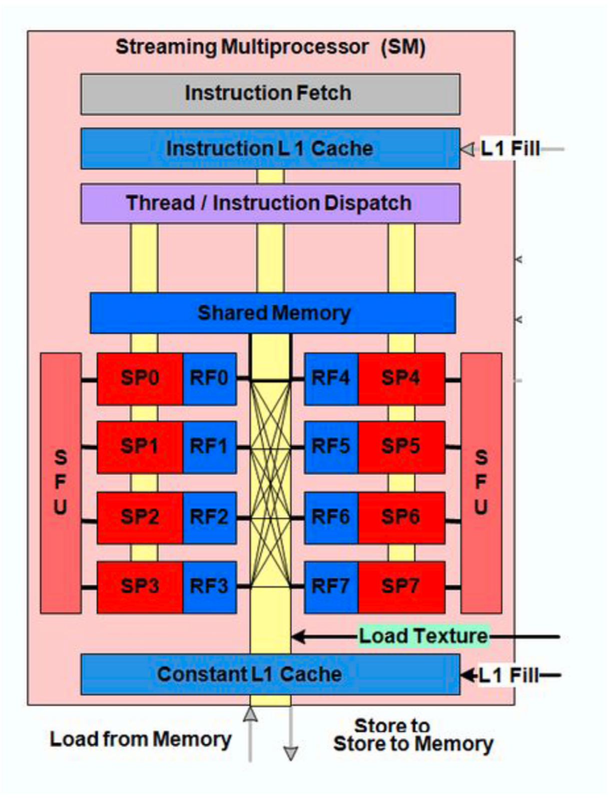

Fig.3. *Structure of the streaming processor*

The usage of GPU for calculations with the help of such graphic program interface was quite possible, but the special approach was necessary. Even item addition of two vectors demanded a drawing in the out-screen buffer. Besides there were much of other restrictions, in fact, the pixel shader is only the formula of dependence of final color of a pixel from its coordinate, and language of pixel shaders - language of recording of these formulas with C-like syntax. It is possible to say that early methods GPGPU were the artful tricks allowing somehow to use the capacity of GPU for the general calculations. The data have been submitted there by pseudo-images (structures), and algorithm-process of a rasterization. Besides it is necessary to note the rather specific model of usage of memory and execution of the program.

3. ADVANTAGES OF CUDA BEFORE THE TRADITIONAL APPROACH TO GPGPU CALCULATIONS

Occurrence of CUDA (and also GPU G80) has completely removed all these restrictions, offering for GPGPU the simple and convenient model. In this model GPU it is examined the specialized computer (named **device**), which:

- is the coprocessor to CPU (named **host**);
- possesses own memory (DRAM);
- possesses an opportunity of parallel performance of huge quantity of separate computing processes **(threads)**;
- more effective data transfer between system and video memory is provided;
- is absent necessities of graphic API with redundancy and overhead charge;
- linear addressing of memory and an opportunity of recording to any addresses are used;
- there is a hardware support of integer and bit operations.

4. HIERARCHY OF COMPUTING PROCESSES (THREADS) IN CUDA

The stream represents a stream of elements of one type which are required to be processed. **Thread** - the process of processing of stream‘s element. All threads are grouped in hierarchy - grid/block/warp/thread (see Fig.2). **The warp** represents a group of 32 streams and is a minimal volume of the data processable in the SIMD-way in multiprocessors CUDA. Threads actually carry out the same commands, but everyone with the data. **The block** in CUDA, it is possible to work with **the blocks** containing from 64 up to 512 streams instead of work with warps directly. All streams of the block are carried out on one multiprocessor **Grid** - blocks are gathered in grids. The advantage of a similar grouping consists of the idea that the number of the blocks simultaneously processable GPU are closely connected to hardware resources. The grouping of blocks in grids allows to abstract from this restriction and apply **a nucleus / kernel** to the greater number of streams for one call, and you can not think about the fixed resources. **Kernel** - the function which will be applied independently to each element of a stream; it is an equivalent of a pixel shader. In classical programming it is possible to result analogy of a cycle - it is applied to the big number of the elements.

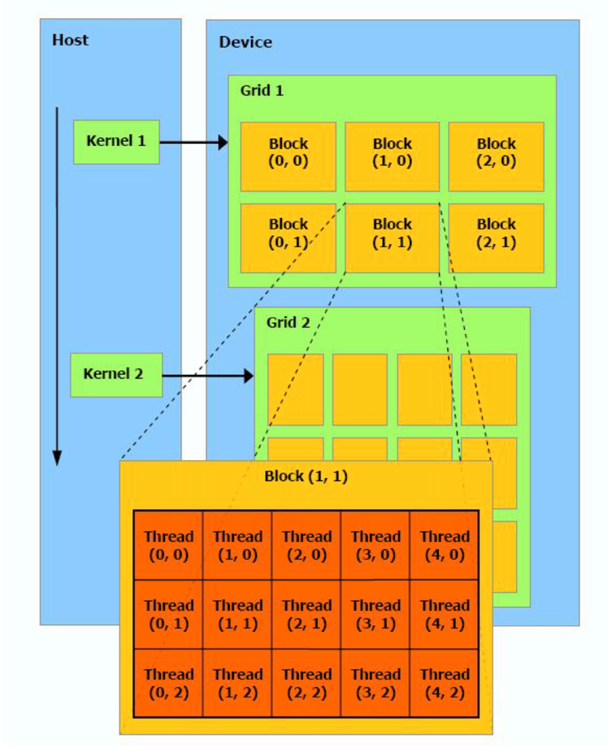

Fig.4. *Hierarchy of the threads in CUDA*

The model of programming in CUDA assumes grouping of the streams. The streams are united in the blocks of streams (thread block) - one-dimensional or bidimentional grids of the streams cooperating among themselves by the means of shared memory and points of synchronization. The program (a nucleus, kernel) is executed above a grid (grid) blocks of streams (thread blocks), see a figure below. One grid is simultaneously executed. Each block can be one, two or three-dimensional under the form, and can consist of 512 streams on the current hardware maintenance. The blocks of streams are carried out as the small groups, named a warp, which size is 32 streams. It is the minimal volume of the data which can be processed in the multiprocessors. And as it is not always convenient, CUDA allows to work with the blocks containing from 64 up to 512 streams. The grouping of blocks in grids allows to avoid the restrictions and apply a nucleus to the greater number of streams for one call. It helps in scaling. If GPU has not enough resources, it will carry out blocks consistently. In a return case, the blocks can be carried out in parallel, that is important for optimum distribution of a work to video chips of a different level, beginning from mobile and integrated.

5. MODEL OF MEMORY CUDA

The model of memory in CUDA differs by the opportunity of byte addressing, supporting both a gather, and a scatter. A plenty of registers on each stream

processor, up to 1024 pieces are accessible enough. The access to them is very fast and you can store in them 32-bit whole or numbers with a floating point [6],[7],[8]. Each stream has access to the following types of memory:
Global memory - makes the great volume of memory accessible to all multiprocessors on the video chip, the size from 256 megabytes up to 1.5 gigabytes on the current decisions (and up to 4 Gbytes on Tesla). It possesses a high bandwidth, more than 100 gigabyte / second for top decisions NVIDIA, but has very big delays in some hundreds steps. It is not cached, supports the generalized instructions load and store, and usual indexes for memory.
Local memory is a small memory size to which only one stream processor has access. It is rather slow - similar to global memory.
Shared memory is 16-kbite (in video chips of present architecture) block of memory with the common access for the all streaming processors in the multiprocessor. This memory rather fast, like, registers. It provides interaction of streams, copes the developer directly and has low delays. The advantages of shared memory are the usage as a cache of the first level controlled by the programmer, reduction in delays at access of executive blocks (ALU) to the data, reduction of quantity of manipulations to global memory.
Memory of constants - the area of memory in volume of 64 kilobytes (the same - for present GPU), accessible only for reading by all multiprocessors. It is cached on 8 kilobyte on each multiprocessor. It is rather slow: a delay in some hundreds steps at absence of the necessary data in a cache.
Textural memory - the block of memory accessible to reading by all multiprocessors. Selection of the data is carried out by the means of textural blocks of the video chip: the opportunities of linear interpolation of the data without additional expenses are given. It is cached on 8 kilobyte on each multiprocessor. It is slow like a global memory - hundreds steps of a delay at absence of the data in a cache. Naturally, the global, local, textural memories and memory of constants are physically the same memory known as a local video memory of the video adapter. Their differences are in various algorithms of caching and models of access. The central processor can update and requests only external memory: global, constant and textural.

6. EXPANSIONS OF LANGUAGE C

The peculiarities of usage of architecture GPU for usual calculations have found the reflection in rather small expansions of language:
Specifiers of the functions were entered showing where function will be carried out and from where it can be called `(_device_, _global_, _host_)`.
The specifiers of variables are supported by specifying the type of memory and using for given variables `(_device_, _constant_ and _shared_)`.
The built variables are added in language containing runtime the information on the current thread:

- gridDim - the size of grid (has type dim3);
- blockDim - the size of the block (has type dim3);
- blockIdx - an index of the current block in grid (has type uint3);
- threadIdx - an index of the current thread in the block (has type uint3);
- warpSize - the size of warp (has type int).

There are **additional types of the data**: are added 1/2/3/4-dimensional a vector from base types - char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float2, and double2.
The directive of a call of a nucleus. For start of a nucleus on GPU is used the following design construction:
`kernelName <<<Dg, Db, Ns, S>>>` (args)
Synchronization of all threads of the block. For this purpose function **_syncthreads** was added in language "C", management from it will be return only when all threads of the given block will call this function. This function is very convenient for the organization of frictionless work with shared-memory. The most simple example of the usage CUDA will be a simple increase in each element of one-dimensional file at unit (the program "incr.cu"), showing the basic working methods with it.

```
*include <stdio.h> __ global __ void incKernel(float *data) { int
idx=blockIdx.x * blockDim.x +
    threadIdx.x;
data [idx]=data[idx]+1.0f; } int main (int argc,char *argv[]) {
int n = 16*1024*1024; int numBytes = n*sizeof (float);
// allocate host memory
float *a = new float[n]; for(int i=0;i<n;i++)
    a[i]=0.0f;
// allocate device memory
float *dev = NULL; cudaMalloc((void **)*dev,numBytes);
// set kernel launch configuration
dim3 threads = dim3(512,1); dim3 blocks = dim3(n/threads.x,1);
// create cuda event handles
cudaEvent_t start,stop; float gpuTime = 0.0f; cudaEventCreate
(*start); cudaEventCreate (*stop);
// asynchronously issue work to the GPU (all to stream 0)
cudaEventRecord (start,0); cudaMemcpy (dev,a,numBytes,
cudaMemcpyHostToDevice); incKernel <<<blocks,threads>>> (dev);
cudaMemcpy(a,dev,numBytes,
    cudaMemcpyDeviceToHost);
cudaEventRecord (stop, 0); cudaEventSynchronize (stop);
cudaEventElapsedTime (*gpuTime,start,
    stop);
// print the cpu and gpu times
printf (" time spent executing by the
 GPU: % .2f millseconds\n ", gpuTime);
// check the output for correctness
printf ("----------------------\n"); for (int i=0;i <n;i++)
    if(a[i]!=1.0f)
    {
    printf("Error in pos %d, %f\n ",i,
     a[i]);
    break;
    }
// release resources
cudaEventDestroy(start); cudaEventDestroy(stop); cudaFree(dev);
delete a; return 0;
}// End
```

The nucleus is easy arranged - to each thread corresponds one thread, blocks and grid are one-dimensional. A nucleus (function incKernel) on input receives only the index on a file with the data in global memory. The task of a kernel - on threadIdx and blockIdx is to determine which element corresponds to the given thread and increase it. As both blocks and grid are one-dimensional, the number of the thread will be determined as a number of the block multiplied (increased) on quantity of threads in the block, plus a number of a thread inside the block, i.e. "blockIdx.x * blockDim.x + threadIdx.x ". The function "main" is a little bit more complex.It should prepare a file with the data in the memory CPU. After that it is necessary to allocate the memory with the help of cudaMalloc under a copy of a file with the data in global memory (DRAM GPU). Then the data are copied by the function cudaMemcpy from the memory CPU to the global memory GPU. After ending of the copy of the data in global memory it is possible to start a nucleus for data processing and after its call to copy results of calculations back from global memory GPU in memory CPU. Also in this example gauging spent on copying and calculations of time is made and the correctness of the received result is checked. After all selected memory is released.

7. RECEIVING OF THE INFORMATION ABOUT AVAILABLE GPU AND THEIR OPPORTUNITIES

The initial text of one simple program is given below which listing all accessible GPU and their basic opportunities.

```
*include <stdio.h> int main (int argc, char * argv []) {
    int deviceCount;
    cudaDeviceProp devProp;
    cudaGetDeviceCount (*deviceCount);
    printf (" Found %d devices\n ", deviceCount);
    for (int device = 0; device <deviceCount; device ++)
    {
        cudaGetDeviceProperties (*devProp, device);
        printf (" Device %d\n ", device);
        printf (" Compute capability: %d. % d\n ", devProp.major, devProp.minor);
        printf (" Name: %s\n ", devProp.name);
        printf (" Total Global Memory: %d\n ", devProp.totalGlobalMem);
        printf (" Shared memory per block: %d\n ", devProp.sharedMemPerBlock);
        printf (" Registers per block: %d\n ", devProp.regsPerBlock);
        printf (" Warp size: %d\n ", devProp.warpSize);
        printf (" Max threads per block: %d\n ", devProp.maxThreadsPerBlock);
        printf (" Total constant memory: %d\n ", devProp.totalConstMem);
    }
     return 0;
}
```

From the resulted examples it is visible that programs are written on "expanded" C, thus their "parallel part" (kernel) is carried out on GPU and the usual part is carried on CPU. CUDA automatically carries out the division of parts and management of their start.

8. OPTIMIZATION OF PROGRAMS ON CUDA

Naturally, within the framework of this article it is impossible to consider serious questions of optimization in CUDA programming, therefore in brief we shall mention about base things. For the effective usage of the opportunities of CUDA it is necessary to forget about usual methods of spelling of programs for CPU and use those algorithms for which the multisequencing on thousand streams is well carried out. Also it is important to find an optimum place for a data storage (the registers, shared memory, etc.) to minimize data transfer between CPU and GPU and use buffering[7]. Within optimization of program of CUDA it is necessary to try achieving optimum balance between the size and quantity of blocks. A lot of streams in the block will reduce the influence of delays of memory, but also will bring down the accessible number of registers. Besides, the block of 512 streams is inefficient; NVIDIA recommends to use the blocks on 128 or 256 streams as a compromise value for achievement of optimum delays and quantities of registers. The basic moments of the optimization of programs CUDA are more as possible to use an active shared memory because it is faster than global video memory of the video adapter; operations of reading and recording from global memory should be incorporated (coalesced) as far as possible. For this purpose it is necessary to use special types of the data for reading and recordings, at once on 32/64/128 bat given by one operation. If the operations of reading are difficult to unite it is possible to try using textural samples.[8] It is necessary to take into consideration one more useful feature of CUDA 2.0 which, however, has not the attitude to GPU - the compiler now provides compilation code CUDA in a highly effective multiline SSE code for fast execution on the central processor. Now this opportunity suits not only debugging, but also real usage on systems without the video adapter NVIDIA. In fact the usage of CUDA in the usual code restrains by the fact that the video adapters NVIDIA are though the most popular, but are not available in all systems. And up to the version 2.0 in such cases it should do two different codes: for CUDA and separately for CPU. And now it is possible to carry out any CUDA program on CPU with the high efficiency, however with smaller speed than on GPU.

9. SUMMARY

The usage of CUDA in the server center of NSC KIPT has shown that though labour input of programming GPU with help of CUDA is rather big, it is much lower than with early GPGPU decisions. The software of SDK CUDA is established and enough steadily works without special problems. However CUDA programming for each multiprocessor is similar of OpenMP programming, It demands a good understanding of the organization of memory. But, certainly, the complexity of development and carring on CUDA strongly depends on the application. It is necessary to get used to other paradigm of programming inherent in parallel calculations. Such programs demand splitting the application between several multiprocessors like MPI programming but without division of the data which are stored in the common video memory. CUDA enables to the developer an opportunity to organize at own discretion access to a set of instructions of the graphic accelerator and operate its memory, organize on it the complex parallel calculations. The graphic accelerator with supporting CUDA becomes a powerful programmed open architecture like today's central processors. All these gives the high level, controlled and high-speed access to the equipment at disposal of the developer, doing CUDA an effective basis for construction of the serious applications.

References

1. A.Zubinsky. NVIDIA CUDA: graphics and calculations unification, (http://itc.ua/node/27969).
2. D. Luebke. Graphics CPU-not only for graphics, (http://www.osp.ru/os/2007/02/4106864/).
3. D. Luebke, G. Humphreys. How GPUs Work// *IEEE Computer, February 2007.* IEEE Computer Society.
4. A.V. Boreskoff.Bases CUDA, (http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html/).
5. A.V. Boreskoff. Bases of programming GPU, stream model of calculations, realization of conditional operators on modern

GPU, (http://steps3d.narod.ru/tutorials/gpu-programming-tutorial.html).

6. D. Chekanov. NVIDIA CUDA: calculations on the videoadapter or death CPU? (http://www.thg.ru/graphic/nvidia_cuda/one page.html).

7. A. Berillo. NVIDIA CUDA - not graphic calculations on graphic processors, (http://www.ixbt.com/video3/cuda-1.shtml).

8. D. Chekanov. NVIDIA GeForce GTX 260 and 280: new generation of videoadapters, (http://www.thg.ru/graphic/geforce_gtx_260_280/geforce_gtx_260_280-02.html).

9. I. Oskolkov. NVIDIA CUDA - the accessible ticket in the world of the big calculations, (http://www.computerra.ru/interactive/423392/).

ПРИМЕНЕНИЕ ВОЗМОЖНОСТЕЙ ИНСТРУМЕНТАЛЬНОЙ СИСТЕМЫ "CUDA"ДЛЯ ПРОГРАММИРОВАНИЯ ГРАФИЧЕСКИХ ПРОЦЕССОРОВ В НАУЧНО-ТЕХНИЧЕСКИХ РАСЧЁТАХ

В.А. Дудник, В.И. Кудрявцев, Т.М. Середа, С.А. Ус, М.В. Шестаков

Описаны возможности технологии "CUDA"(Compute Unified Device Architecture – унифицированного программно-аппаратного решения для параллельных вычислений на GPU) компании NVIDIA. Выделены основные отличия языка программирования С для GPU от "обычного"С. Даны примеры использования CUDA для ускорения разработки приложений и реализации алгоритмов научно-технических расчётов, выполняемых средствами графических процессоров (GPGPU) ускорителей GeForce восьмого поколения. Приведены рекомендации по оптимизации программ, использующих GPU.

ЗАСТОСУВАННЯ МОЖЛИВОСТЕЙ ІНСТРУМЕНТАЛЬНОЇ СИСТЕМИ "CUDA"ДЛЯ ПРОГРАМУВАННЯ ГРАФІЧНИХ ПРОЦЕСОРІВ В НАУКОВО-ТЕХНІЧНИХ РОЗРАХУНКІВ

В.О. Дудник, В.І. Кудрявцев, Т.М. Середа, С.О. Ус, М.В. Шестаков

Описано можливості технології "CUDA"(Compute Unified Device Architecture – уніфікованого програмно-апаратного рішення для паралельних обчислень на GPU) компанії NVIDIA. Виділено основні відмінності мови програмування С для GPU від "звичайного"С. Дани приклади використання CUDA для прискорення розробки і реалізації алгоритмів науково-технічних розрахунків, виконуваних засобами графічних процесорів (GPGPU) прискорювачів GeForce восьмого покоління. Приведено рекомендації по оптимізації програм, використовуючих GPU.