Educational Aspects of Massively Parallel Program Development

Éva Hajnal
Óbuda University, Alba Regia Technical Faculty, Szekesfehervar, Hungary
hajnal.eva@arek.uni-obuda.hu

Abstract **– The last three years began the education of massively parallel programming at our faculty. This means regular contact education consisting of lectures and computer laboratory practice with computers containing simple CUDA capable graphics cards (NVIDIA GeForce GT 430) and supervising student's projects. This paper summarizes the educational analysis of two projects and gives suggestions of the future direction and methods of massively pareallel programing education.**

I. Introduction

The milestone of GPGPU programming was the year of 2004, when DirectX 8, a new graphics pipeline was developed, by which programmable vertex and pixel shader layers can be defined. In this system, the instructions were programmable with a special language, HLSL, which was developed directly for this purpose. If the data were embedded into a special graphics data structure, it was capable to manipulate on them. This method was rather circumstantial, but the reason for existence of it was in special graphical and scientific applications [1].

The most important innovation of DirectX 10 was the totally programmable graphics pipeline, which is mentioned in the literature as unified pipeline architecture. The prior fix pipeline becomes programmable with HLSL, so the dedicated processors become unnecessary. Developments of hardware manufacturers make possible to solve more general programmable problems with these devices and passing over the graphic API's, using GPU for general purpose computation. The hardware manufacturers created new independent programming interface for GPU's and appropriate compilers, which are capable to compile a higher level of GPU programming language then the HLSL onto the reduced instruction set architecture. Such interface is the Compute Unified Device Architecture (CUDA), which appeared in the year of 2008 and till now it is the most popular solution. At the same time the Khronos Group developed OpenCL and a bit later Microsoft connected to them with C^{++} Amp.

Now there is a large competition in this sector as well, but by the forecasts the OpenCL or CUDA will be the determinant in the future.

CUDA is the programming platform of NVIDIA which is programmed with C and Fortran language extensions. The vendors offer a complete development kit, CUDA SDK, which contains C and Fortran compilers and function libraries to facilitate the software development process. The NVIDIA Parallel Nsight is an additional tool for software debugging.

During the appearance of these tools, there were developed a series of products to help the computation on massively parallel processors, such as function libraries in different topics and wrappers. Since 2004 the Parallel Computing Toolbox for Matlab is under continuous development and now, among others, contains instructions for GPU's. These indicate that the parallel algorithms became useful in the everyday technical and computational practice and in user level we can also operate with them.

In the same time there was a demand for developing software on higher level languages and for comfortable execution framework, such as C#, Visual Basic and for .NET, and among others the CUDAFY SDK [2] play such role since 2012 helping us to become acquainted with CUDA.

The education of GPU programming started from 2010/11 as optional subject. The hardware devices made possible to evaluate small projects in the first semester, and later, from the year of 2012/13 several computers were extended with NVIDIA GeForce GT 430 cards, and from that time systematic education with weekly lectures and computer laboratory practices were realized. The education is matching to the other subjects uses Cudafy SDK and C# programming language in Microsoft Visual Studio environment.

However this subject is quite popular, the programming of massively parallel processors does not seem to be successful from lots of aspects at us and also over the world.

In this paper the analysis of two student's project is described, and the advantages and disadvantages of this discipline are investigated from educational aspect. These projects are examined from the aspect of program development, testing and optimization in the context of the software performance and human time consuming.

II. Project I.- Cellular Automaton Implementation in GPU System

The task was implementing a cell automaton on CPU and GPU, and comparing the execution time of the program, and the time interval of software development [3–5].

The original problem was developing an ecological model for phytoplankton communities of which species

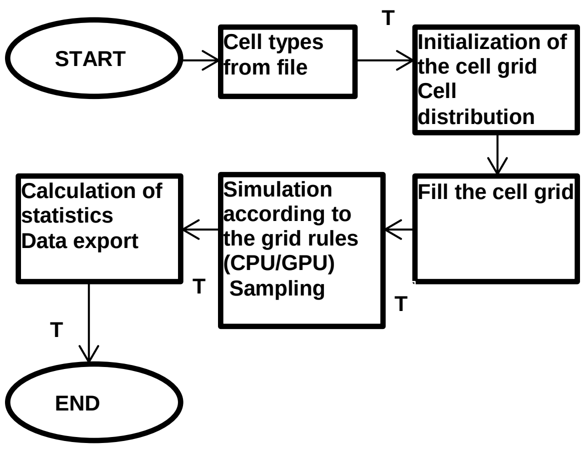

START
Cell types from file
T
Initialization of the cell grid Cell distribution
Fill the cell grid
T
Simulation according to the grid rules (CPU/GPU) Sampling
T
Calculation of statistics Data export
T
END

Figure 1. Block diagram of the cell automaton software, T means the time check points in the source code.

coexisting with neutral interaction. This software can simulate the stochastic behavior of this ecosystem, where the number of species, the mean reproduction and mortality rates of each species, the maximum lifetime is readable from a file. The execution of the simulation is facilitated with the parallelism of CUDA technology if the computer contains GPU card (Fig 1).

The software can be executed on Windows operating system, and a MSSQL Express database server needs for the storage of simulation results. The CUDA capable graphics card does not required only recommended, because the simulation may be executed on CPU, but the execution time can be extraordinary large. This application does not contain graphics user interface, but the settings are the input parameters.

According to the nature of GPU programming, the software development was strongly connected to the hardware solution. In this case the developer configuration based on an AMD Phenom II X4 processor with 4 cores, supplemented with 4 GB RAM and Asus ENGTX460 TOP video card, which contains a G104 GPU (7 stream processor and 32 CUDA core/ processor).

By the point of view of NVIDIA, the optimal usage of the graphics card is the responsibility of the software developers, so they have to calculate the number of threads, and how many bytes from memory is occupied. This is very important question, because there is no way to free the allocated memory (included the free instruction in the source code) until the end of the kernel execution. If this problem is not handled, the kernel can allocate the whole memory capacity of the graphics card, and the calculations that concerning the displayed image do not executed, which leads to an operating system error [1], [6], [7]. There are lots of methods for the calculation of the number of threads; in this project the CUDA Occupancy Calculator [8], a simple Excel spreadsheet of NVIDIA was used.

Because of the initial stage of this subject, the decision about the software development environment was not so easy, and it was followed by the actuation of this system.

At this project the Visual Studio 2010 Professional version and CUDA 3 version were chosen, because it is matching to our mainly used configuration. Visual Studio builds the projects by a project template file, but there were no CUDA templates for any graphics environment. Finally the creation, edition of the template became the part of this project. Additionally we had to setup the Visual Studio 2008 C++ Express version also, because the compiler of this can work on a CUDA project, and its intellisense tool can at least limitedly helps in this project. The syntactic check does not cooperates with CUDA, because it's syntactic elements e.g. "<<< "and ">>>", that are the kernel call, are unknown for it. These features are partly exist nowadays, and can enhance the duration of the software development.

The software was implemented by the object oriented paradigm. The defined classes were grouped by scope of duties, so they are different by function. For the help of the later enrichment, base classes as prototypes of some features were developed, and the concrete implementations were placed into the derived classes. By this solution there are several implementations can be made, and can be replaced without any other changes in the source code. This method concerned species distributions and grid rules classes. The ***Idistribution*** base class and its derived classes fill the cell grid according to different distribution function (lognormal, zipf etc.). The ***Rule*** base class also a re-definable class, which implements the cell grid rule system.

Further classes in the software are the ***Cell*** and ***Coord*** structures, the ***Cell factory*** class, which deals with the cell creation and ***Cell-space*** class, which is responsible for the cell grid manipulations. Important feature of CUDA projects, that the GPU can manipulate effectively on one dimensional arrays, and consequently there is a continuous demand on the conversion between the several dimensional arrays and vectors on logical or on data level. The statistics is handled by the ***Results*** and ***Resultshandler*** classes, which are implemented on CPU.

For the measurement and comparison of GPU and CPU execution time check points were settled into the source code. These points are between the source code modules, so the creation of the cell grid, the simulation of the cell automaton and the calculation of the statistics can be measured on CPU and GPU (Table 1).

TABLE 1. PERFORMANCE COMPARISON OF CPU AND GPU PROGRAM EXECUTION

Number of cpecies	**10**							
Number of iterations	**1000**							
Number of cells	1024		4096		16384		65536	
	CUDA	CPU	CUDA	CPU	CUDA	CPU	CUDA	CPU
Cellspace Creation (ms)	47	47	187	187	764	749	3026	2995
Execution of the Simulation (ms)	1092	3885	1154	7613	1529	19454	1820	198262
Data export (ms)	78	78	78	78	78	78	78	78
Total (ms)	1217	4010	1419	7878	2371	20281	4924	201335

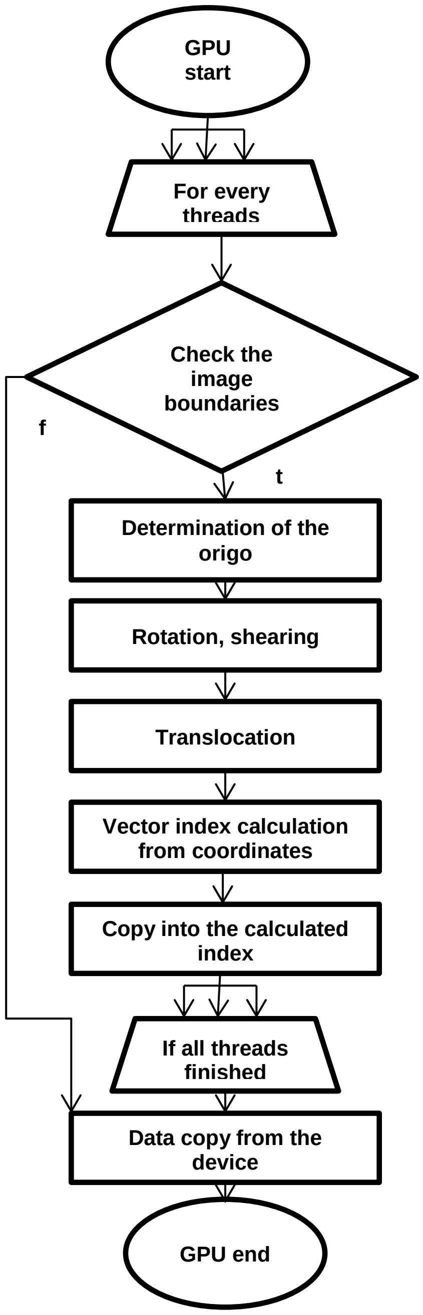

GPU start
For every threads
Check the image boundaries
f
t
Determination of the origo
Rotation, shearing
Translocation
Vector index calculation from coordinates
Copy into the calculated index
If all threads finished
Data copy from the device
GPU end

Figure 2 Block diagram of the software

Our hardware possibilities at that time did not make possible the execution of debugging and profiling tools, so the optimization of the source code was evaluated in small-scale. Mainly theoretical considerations helped us to stop or reduce the factors, which made slowly the data parallel code execution (SIMD programming model), such as thread divergence etc. In spite of it, the execution time of this software on GPU was significantly shorter, often, especially at the really used cases, could be reached 1-2 order of magnitude speedup to the CPU code (Table 1).

Unfortunately the duration of the software development and testing without substantial optimization was about 2-3 fold to the desktop CPU software development.

III. Project II. – Real time affin transformations on images

In the industrial practice raised the request for a real time image processing program library, which can recognize and measure some products on the conveyor belt independently from their position. This purpose can be reached by several 2D geometrical transformations such as shearing, rotation and translation. These transformations are called affine transformations, and their mathematical basis is solved by matrix manipulations [9], [10]. Figure 3 shows the integrate description of these calculations with homogenous coordinates. For 2D transformations a special 3x3 matrix can be used, of which last row is constant (1), and needs only for the symmetry of operations.

Every pixel has a coordinate (x, y), in which the x means the column and y means the row. The modification of the pixel coordinates results the image manipulations, rotation, translation, reflexing etc.

However the theoretical basis of this method is simple, the computational demand is high at large resolution images. The problem suggests the data parallel solution.

This was made on CUDA compatible GPU card which compute capability is minimally 2.1. The software development was realized on Nvidia Geforce GT 440 card, and optimized onto Windows 8 operating system, and :NET 3.5 framework with CUDA driver. A language of the host program was the C#, and the Visual Studio 2010 Professional version was the development environment, which was supplemented with a CUDA wrapper, CUDAfy 1.22 version [2].

Nvidia Visual Profiler and Nvidia Nsigh were used for the software test and analysis. They can integrate into the Visual Studio, which make the work easier.

The real time software debugging is not possible in our laboratory configuration, because the GPU card can execute only one task, either the calculations or the debugging. For the real time debug needs a second GPU card.

The principle of the solution is a map back algorithm, which was programmed both on CPU and GPU. (GPU block diagram is on fig 2).

Optimization possibilities were taken into account when the program was developed. Especially the memory usage was restricted. In the present technology we can use only 32 byte register capacity per thread. The purpose is to avoid the extra bytes, which are copied into the much more slowly shared memory.

Other optimization technics are the instruction level optimization with optimal instructions (shift >>1 instead of divide by 2) and the reducing of the number of instructions (intermediate data storage into a register). The code insurance supported with quick execution is possible with the usage of atomic instructions.

The main factor in the execution speed is the determination of the optimal number of blocks and threads in the grid, during the kernel code launch.

$$\begin{bmatrix} a & b & u \\ c & d & v \\ tx & ty & w \end{bmatrix}$$

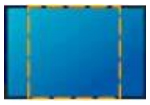
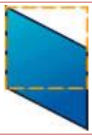
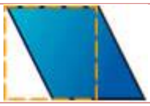
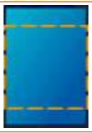
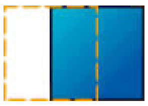
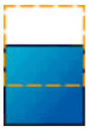
a:	enlargement in x direction	
b:	shearing in y direction	
u:	constant, 0	
c:	shearing in x direction	
d:	enlargement in y direction	
v:	constant, 0	
tx:	translation by x axis	
ty:	translation by y axis	
w:	constant, 1	

Figure 3 The matrix of the coordinate transformation, and the meaning of the elements of this matrix.

The warp is not an architectural part of the GPU, but a scheduling unit. Threads of one warp execute the same operation parallel, so the threads inside one warp do not need synchronization. The more fully allocated warps are working the more efficient is the data processing.

However there is no need for synchronization, sometime a thread and consequently a whole warp does not work. This can occur, if the thread is waiting for a memory word to an operation, but the memory access needs relatively long time on the graphics card. When a warp is idle, the timer start another ready warp. If there are several ready warps, the choice is made by priority.

If the number of blocks and threads is not suit the card's possibilities, the number of warps will be higher and the number of threads in a warp will be lower than the optimum, that causes significant slow, and in extreme cases the timer cannot execute any warps.

The optimization of the warps does not cause speedup in all cases, because the instructions in the kernel code also may cause slow. If there are not so much memory read or write operations, the bottleneck of the execution is the warp scheduling, but in other cases the execution time is limited by the memory bandwidth. According to the general experience, the optimization of memory usage is a serious and frequently occurring need in GPU programming.

After the program development and optimization several software execution parameter were measured and the CPU and GPU execution were compared (Table 2). The purpose of the project was reached, because a relatively cheap and low compute capability graphics card could realize real time calculations on about 1000x1000 pixel resolution images. However the program development environment was much more maturated and its possibilities made the work more comfortable and effective the duration of the software development onto GPU remained at least 4-8 fold to that one onto CPU.

TABLE 2 COMPARISON OF THE PROGRAM EXECUTION ON GPU AND CPU

Image size (pixel)	Data (Byte)	CPU execution (ms)	GPU execution (ms)	Compute capacity of GPU (GFLOPS)
40x134	3331	0.69	0.88	4.77
2048x1536	872373	471	19	9.56
3648x2736	3824694	1515	168	4.98

IV. DISCUSSION

The parallel programming concept was raised at the first time of the computing technic, and till now there were made a lot of efforts to implement it. Nowadays we can say that it is the part of the everyday software technological practice. It became useful not only in specific tasks or in large computing systems, but in almost all the desktop applications, which can be used on personal computers. Especially the data parallel programming concept is used widely. So the teaching and learning of this subject is evident in the education of engineers as it was realized in all faculty of informatics or technical faculty.

The human brain originally can work sequent or very often can execute parallel tasks, but the data parallel thinking - which seems to be very simple at the first meet - is proved to be alien for us. The consequence of it is very serious. However there is a continuous ambition to develop the services of the data parallel environment making the work more and more comfortable. The duration and the effort in the software development processes both in the education and in the industry remain relatively high, which keeps the costs of the software development at high level.

In the future we can meet with the reduction of the software development costs by the creation of widely useful, general data parallel program libraries. It will cause some performance lost because of the missing of the specific optimization. Finally there are nowadays tendencies to move the decision of the usage of data parallel functions onto the hardware optimization level. These tendencies are very important aspects in the education, which requires teaching the data parallel principles, algorithms and the methods of embedding data parallel functions into the applications, and makes optimization methods a bit subsidiary.

V. Acknowledgment

Thanks for Tamás Bajzát and Dávid Horváth for motivating me to continuous learning.

The project was supported by TÁMOP-4.2.3.-12/1/KONV „Alba Regia Egyetemi Központ tudományos eredményeinek disszeminációja, mérnöki és kutatói utánpótlás biztosítása a közép-dunántúli régióban” grant.

VI. References

[1] D. B. Kirk and H. Wen-mei W., *Programming Massively Parallel Processors, Second Edition*, Elsevier. Waltham USA: , 2012.

[2] “Cudafy user Manual 1.12,” 2012.

[3] T. Bajzát, “Cell Automaton Modelling Algorithms: Implementation and Testing in GPU Systems,” in *15th International Conference on Intelligent Engineering Systems*, A. Szakál, Ed. Budapest: IEEE Hungary Section, 2011, pp. 177–181.

[4] É. Hajnal and T. Bajzát, “Ecological Modelling with Cellular Automaton Software Implemented in GPU System,” *ÓBUDA UNIVERSITY E-BULLETIN*, vol. 2, pp. 499–508 PG – 9, 2011.

[5] É. Hajnal and T. Bajzát, “Parallel Programming in GPU Systems: Case Study,” in *International Symposium on Applied Informatics and Related Areas*, Székesfehérvár: Óbudai Egyetem, 2011, p. &.

[6] R. Farber, *CUDA application design and development*, Elsevier. Waltham USA: , 2011.

[7] E. Kandrot and J. Sanders, *CUDA by Example*. Michigan USA: , 2011.

[8] M. Harris, “CUDA Occupancy Calculator.” [Online]. Available: https://devtalk.nvidia.com/default/topic/368105/cuda-occupancy-calculator-helps-pick-optimal-thread-block-size/.

[9] L. Szirmay-Kalos, *Számítógépes Grafika*. Budapest: , 1999.

[10] L. Szirmay-Kalos, “Számítógépes grafika,” *Firka*, 2009.