GPU Optimized Computation of Stencil Based Algorithms

L.M. Itu, C. Suciu, F. Moldoveanu, A.Postelnicu
Department of Automatics
Transilvania Univeristy of Braşov
Braşov, Romania
lucian.itu@unitbv.ro, suciuc@unitbv.ro, moldof@unitbv.ro, adip@unitbv.ro

C. Suciu
Corporate Technology
PSE Siemens Romania
Braşov, Romania
constantin.suciu@siemens.com

Abstract — **The paper describes an optimized GPU based approach for stencil based algorithms. The simulations have been performed for a two dimensional steady state heat conduction problem, which has been solved through the red black point successive over relaxation method. Two kernels have been developed and their performance has been greatly improved through coalesced memory accesses and special shared memory approaches. The approach described in the paper does not only represent a step forward for the steady state heat conduction problem but also for any other algorithm which performs the numerical solution of partial differential equations or which is stencil based. The paper not only describes the various code versions but also the process which has lead to these improvements. Also the optimized GPU code version has been compared with the corresponding CPU version. The testing results show that the GPU algorithm always leads to an improvement. The value of the improvement though greatly depends on the number of grid points on which the computations are performed.**

Keywords: *GPU, heat conduction, shared memory, optimization, successive over relaxation, speed-up*

I. INTRODUCTION

Graphics Processing Unit (GPU) based implementations have introduced an alternative to CPU based solutions. GPUs were initially used only as graphical accelerators in image processing applications. A GPU is a many-core processor, which, given the need of the graphical applications, is designed in order to execute a large number of floating point operations in parallel on hundreds of cores [1]. The latest GPU architecture performs at over 1 teraflop, thus being several times faster than multi-core CPUs. The transition from graphics applications to general purpose applications has been made possible by the introduction of CUDA (Compute Unified Device Architecture) [2].

When a GPU is programmed through CUDA, it is viewed as a compute device, which is able to run thousands of threads in parallel by launching a kernel (a function, written in C language, which is executed by the threads on the GPU) [3]. The latest GPUs contain several streaming multiprocessors, each of them containing eight cores.

Currently all applications which use a GPU in order to accelerate the execution also use the CPU in order to perform auxiliary tasks (like initializations or post-processing) and also to launch the kernels [4]. Until now the GPU is not able to run as a stand-alone device, it needs to be launched by a host thread which also manages the data located in the global memory (it copies initial values to the device and copies the results back at the end of the execution).

A major parameter, indicating whether it is worth to move the computational intensive part of a program to a GPU, is the execution time compared to the situation when the code is executed entirely on the host. In order to have a complete picture of the comparison, when the execution time for the GPU is determined, the execution times required by the memory copies between host and device have to be taken into consideration.

One of the most important scientific fields which has profited from the general purpose character of the GPU is computational fluid dynamics (CFD). Leading CFD experts have immediately identified the potential of the GPU to accelerate their applications, when the CUDA language was announced [5]. Up to date, GPU based applications can already compete with CPU based supercomputers and multi-GPU applications are able to increase the level of performance even further. As with all parallel applications the goal is to execute the same application in less time, to execute a more complex application in the same time or a combination of both.

The paper introduces a novel approach regarding the usage of shared memory in order to improve the computation time of stencil based algorithms on GPUs. The technique has been applied for the solution of a two dimensional steady heat conduction problem. The numerical solution is based on the finite difference method (FDM) [6] together with the red-black Gauss-Seidel method, which iteratively solves the above mentioned elliptic partial differential equation.

Section two provides a detailed description of the heat conduction problem and of the numerical methods which can be used to solve it. Section three focuses on the GPU implementation and on the migration of the computationally intensive part of the program to the GPU. Section four describes the code optimization activities. Section five presents

the results of the research activity and shows the performance improvements brought by the GPU version. Finally we will draw some conclusions on our work in chapter five.

II. The Steady State Heat Conduction Problem

The steady state heat conduction problem represents an elliptic partial differential equation [7]. Elliptic equations are a fundamental building block in fluid dynamics. In case of elliptic equations, the value of a certain grid node in the xy-plane influences all other grid nodes of the domain [8]. For the other two types of partial differential equations (hyperbolic and parabolic ones) the value at a grid node influences only a part of the rest of grid nodes. The numerical algorithms which can be used are into two main categories [9]:

- direct methods (Gaussian elimination, the Thomas algorithm or Chelosky methods);
- iterative methods (Jacobi, SOR, ADI, etc.).

The equation, which will be used in the following is:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \tag{1}$$

We consider a five point finite difference scheme, which leads to the following discretized formula:

$$\frac{T_{i+1,j} - 2 \cdot T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2 \cdot T_{i,j} + T_{i,j-1}}{\Delta y^2} = 0 \tag{2}$$

To accelerate the convergence rate of the algorithm, we have chosen the point successive over relaxation method (PSOR), which leads to the following iterative formula [10]:

$$T_{i,j}^{k+1} = (1-\omega) \cdot T_{i,j}^{k} + \frac{\omega}{2 \cdot (1+\beta^2)} \cdot \left[T_{i+1,j}^{k} + T_{i-1,j}^{k+1} + \beta^2 \cdot (T_{i,j+1}^{k} + T_{i,j-1}^{k+1}) \right] \tag{3}$$

where ω is a relaxation parameter and β depends on the spatial discretization.

The numerical scheme obtained by applying equation (3) is called explicit because it contains a single unknown value [11]. Because the goal of this paper is to describe the GPU aspects of the implementation, we will solve the heat conduction problem on a simple, rectangular domain as the one in fig. 1. The boundary conditions are known on all four boundaries, and the boundaries coincide with the grid lines.

III. GPU Implementation of the Heat Conduction Problem

We will describe in the following the changes which have to be performed in order to allow the parallel execution of eq. (3). On a CPU the nodes are usually computed sequentially in a row major order (or column major order).

In CFD applications a stencil describes or displays the neighboring grid points/variables which are involved in the equation of a certain grid point (eq. (3) will be solved for every point in the grid). The stencil corresponding to eq. (3) will be using two new values and two old values of the neighboring points (in eq (3) two of the neighboring terms have the superscript *k+1* and the other two *k*). This is a reasonable solution for a CPU but it can not be implemented on a GPU because all of the computations have to be sequential.

The PSOR scheme can be changed so as to allow a parallel execution with fewer sequential steps by the introduction of the red-black or checkerboard scheme (fig. 2). The scheme divides the nodes into two groups: red nodes and black nodes [12]. Fig. 2 shows that when red nodes are computed, values of only black nodes are needed and vice-versa. Hence only two sequential steps will be needed to perform one sweep over the nodes of the grid: first all red nodes will updated and then all black nodes will be updated. All computations corresponding to a certain node color will be performed in parallel. Eq. (3) has to be rewritten for the two sequential steps. In the numerical scheme, the red nodes are those which satisfy the condition (i+j)%2==0 (i represents the row number and j the column number). The following formula will be used for these nodes:

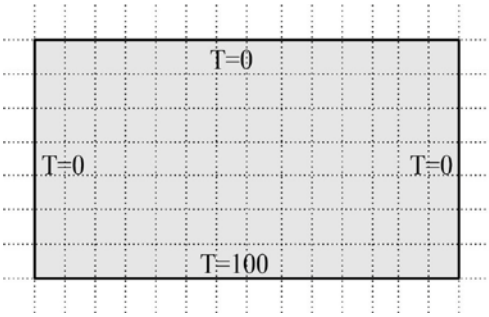

Figure 1. Rectangular domain of the problem and boundary conditions

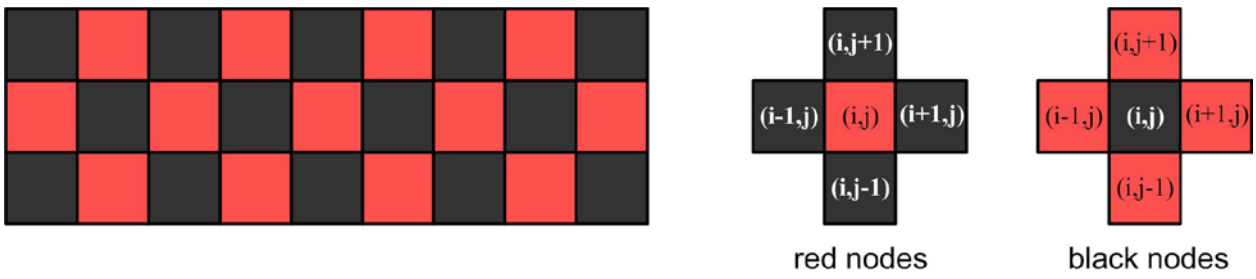

Figure 2. Red black PSOR memory stencils

$$T_{i,j}^{k+1} = (1-\omega)\cdot T_{i,j}^{k} + \frac{\omega}{2\cdot(1+\beta^2)} \left[T_{i+1,j}^{k} + T_{i-1,j}^{k} + \beta^2\cdot(T_{i,j+1}^{k} + T_{i,j-1}^{k})\right] \quad (4)$$

The black nodes will use the following formula:

$$T_{i,j}^{k+1} = (1-\omega)\cdot T_{i,j}^{k} + \frac{\omega}{2\cdot(1+\beta^2)} \left[T_{i+1,j}^{k+1} + T_{i-1,j}^{k+1} + \beta^2\cdot(T_{i,j+1}^{k+1} + T_{i,j-1}^{k+1})\right] \quad (5)$$

At the beginning the host allocates the memory buffers for the host and for the device. In the next step the host memory will be initialized. During this initialization step the values of the nodes which lie on the boundary will be set according the numbers displayed in fig. 1. Because no further information has been considered for the interior nodes of the domain, their values will be initially set to zero. If some better initial values would be available, then the algorithm would converge faster. Since the numerical scheme uses old values while new ones are computed, two different buffers have to be allocated: one for the old values and one for the new ones.

Having initialized the host buffer with the boundary values, the next action is to start the timing of the application in order to compute the exact execution time of the algorithm. This has been performed through the use of CUDA events, which is been considered as the most precise method [13]. Further the host buffer containing the initial values is copied into the device memory in order to prepare the first iteration of the algorithm. The iterations which perform the computation of the grid nodes on the GPU will be executed inside a for loop. As specified earlier, there will be two kernel launches at each iteration, one corresponding to the red nodes and one to the black nodes. When the iterations have been finished and the for loop is exited, the results buffer is copied from the device memory to the host memory. Next the timer is being stopped and the results are post-processed. The execution of the algorithm for the GPU version also includes these copy operations, because these were not necessary for the CPU version.

This is the basic version of the GPU based implementation of the steady state heat conduction problem. The next section will provide a detailed description of the optimization activities which have been performed.

IV. Optimization Activities

During these activities the bottlenecks of the application have been identified. The GPU has several resources, like instruction throughput or memory bandwidth which have to be traded one for each other in order to eliminate the bottlenecks. To be able to determine the bottlenecks, the type of limitation of the algorithm has to be determined. Generally three main causes for the performance limitation can be identified:

- PCI Express Bus limitation: this situation appears when important periods of time are spent on the copies between the host and the device buffers;
- Memory bandwidth limitation: the speed of the algorithm is limited by the latencies which appear at the reading or writing of the global memory;
- Instruction throughput limited: the speed of the algorithm is limited by the number of instructions which can be executed per cycle;

The performances of the algorithms have been measured through the CUDA profiler and the main aspects are displayed in table 1. These values correspond to the execution on a fine grid, on which 100 iterations have been executed.

One can see that only 1.47% of the total execution time is spent on the copies between the host and the device, hence the PCI Express Bus limitation can be clearly ruled out. The value depends on the number of iterations which are executed, thus it will become even lower for a greater number of iterations. Before proceeding to the other two types of limitation, the technical specifications of the GPU have to be taken into consideration. The algorithm has been executed on a NVIDIA Tesla GPU (GTX260), which allows a maximum device memory throughput of 111.9GB/s [14]. The instruction throughput is basically the ratio of the instruction rate which has been achieved and the peak single issue instruction rate. This value depends on the clock speed of the GPU and when dual-issue is considered this figure can exceed the value one. The last two columns of table 1 summarize these two latter aspects, and the values indicate that the memory bandwidth is the main limitation factor

Ref. [15] provides a series of paths to be followed in order to alienate several types of limitation. For the global memory bandwidth limitation there are three main aspects to be considered:

- Global memory accesses should be sequential and aligned (high priority);
- Reduce the usage of global memory, use shared

TABLE I. Table Type Styles

Operation	Nr. of Calls	Execution time [µs]	% of Exec. Time	Global mem. read throughput [GB/s]	Global mem. write throughput [GB/s]	Overall global mem. throughput [GB/s]	Instruction throughput
Black PSOR kernel	100	10702.6	49.78	98.28	13.12	111.4	0.511
Red PSOR kernel	100	10474.3	48.72	100.43	11.27	111.7	0.513
Memcopy H→D	2	85.79	0.39	-	-	-	-
Memcopy D→H	1	232.89	1.08	-	-	-	-

memory instead (high priority);

- Reduce the number of redundant accesses to the global memory through usage of shared memory (medium priority).

These three recommendations will now be considered in order to improve the overall performance of the algorithms.

Sequential and aligned global memory access means that threads lying in the same half-warp read memory locations which are situated inside blocks of 32, 64 or 128 Bytes. The device buffers have been padded in order to fulfill this condition and this action has lead to an increase of 4.2% for the memory usage. The additional time spent on copies between the host and the device is easily compensated by the fact the total number of memory transactions has been reduced. For the kernel computing the red nodes, the memory transactions have been reduced by 23.46% and for the black kernel by 23.69%.

The second recommendation was to use shared memory. Shared memory usage may improve the execution time of GPU algorithms if global memory is accessed redundantly. In case of stencil based computations, this is clearly the case (as fig. 2 displays).

Fig. 3 displays the first type of structure of shared memory which we have tested. One can see that the shared memory block contains an additional block on each side in order to be able to deliver data for all stencils of the block. This shared memory array has lead to a reduction of 42.06% for the memory transactions of the red kernel and of 22.69% for the black kernel. This change though has lead to a modification of the main limitation factor: the application has now become compute limited and the overall performance has been worsened. The application has become compute limited because the total number of divergent branches has increased by over 300%! The red kernel reads all five values of its stencil from the same array, while the black kernel reads the four neighboring values from the array containing the new values and the center value from the array containing the old values (see eq. (4-5)). This is why the total number of memory transaction has decreased much more for the red kernel than for the black kernel.

In order to improve the overall performance of the application a middle way has to be found, which on the one side still reduces the total number of memory transactions but on the other side does not lead to an important increase of the divergent branches. When the central part of the shared

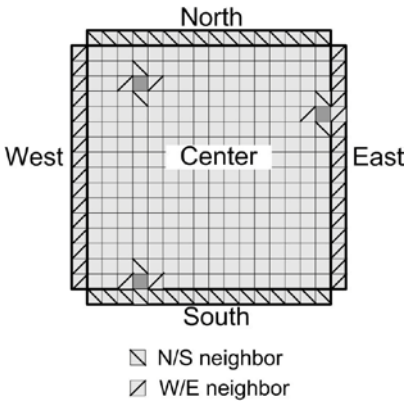

Figure 3. "Full Shared Memory support" pattern

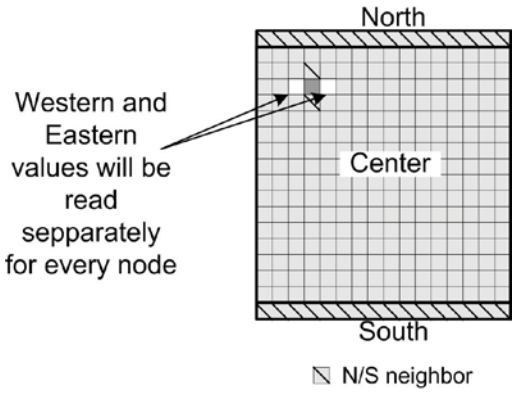

Figure 4. "Partial Shared Memory support" pattern

memory array is read, no divergent branches are introduced. Consequently the four border parts have lead to the decrease in performance. To go further into detail, the warp – half-warp organization of the threads has to be considered. A half-warp will contain 16 threads lying on the same row, hence the reading of the northern and southern block only introduces two divergent branches. The western and eastern parts will lead to one divergent branch for each location, thus causing the high increase in the number of divergent branches. This is why we have decided to eliminate those to sections and to keep only three sections displayed in fig. 4. Thus the western and eastern parts will be read from the global memory and we have named this version "partial shared memory support"

This change has lead to a further improvement of 11% for the execution time of the red kernel and of 5.9% for the black kernel. The decrease in the total number of memory transactions is not as dramatic as for the first shared memory version (33.32% for the red kernel and 14.49% for the black kernel).

V. RESULTS

The iterative improvement activities have lead to a total of four different kernel versions. The aspects which have been discussed throughout the last section are summarized in table 1.

As already specified, the most important performance indicator is the execution time of the algorithm. Fig. 5 summarizes the execution times of the different versions for the red and black kernels. The tests have been conducted on three different grained grids and fig. 6 shows the total improvement which has been achieved for each kernel and for each grid. The

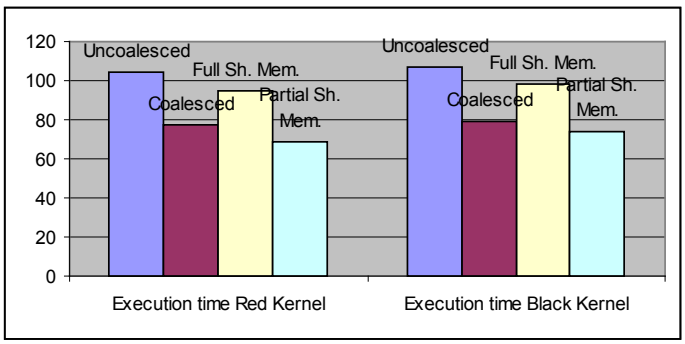

Figure 5. Improvement of the execution time

TABLE II. SUMMARY OF THE FOUR KERNEL VERSIONS

Kernel	Execution time [µs]	Reg. per thread	Divergent branches	32B read ops.	64B read ops.	128B read ops.	Total	GM read through-put	Instruction throughput
Red K. Uncoalesced	104.74	8	402	5674	5659	4941	16274	95.77	0.513
Red K. Coalesced	77.32	8	402	1126	10261	1069	12456	95.77	0.694
Red K. Full Sh. Mem.	94.71	8	1258	4576	2640	0	7216	29.87	1.015
Red K. Partial Sh. Mem.	**68.82**	8	498	1133	6103	1069	8305	73.29	0.918
Black K. Uncoalesced	107.02	8	402	5596	5591	4944	16131	94.11	0.511
Black K. Coalesced	78.95	8	402	1054	10122	1133	12309	94.11	0.692
Black K. Full Sh. Mem.	98.43	9	1258	4592	4924	0	9516	41.91	0.997
Black K. Partial Sh. Mem.	**74.29**	8	498	1069	8331	1125	10525	80.20	0.822

improvements lead to greater increases in performance for the finer grids. The total improvement is of 34.29% for the red kernel and of 30.58% for the black kernel.

Further the changes of the other critical parameters, in terms of performance improvement, are summarized. The changes of the values of the parameters between the initial and optimum versions are:

- 23.88% increase in the total number of divergent branches;
- 48.88% and 34.75% decrease of the total number of memory transactions for the red and black kernel respectively;
- 27.02% and 18.39% decrease of the global memory read throughput for the red and black kernel respectively;
- 78.94% and 60.86% increase of the total instruction throughput for the red and black kernel respectively.

These figures show that the optimization of GPU code is a tedious task which involves several testing activities and usually involves the trading of one resource for another. The values above indicate that three parameters have been improved and only one of them has been worsened. Usually it is not possible to improve all of them because of the complex relations between them. This is why a good start in optimization activities is to try to conduct as many tests as possible.

As specified, the algorithm has been tested on three different grained grids and table 3 summarizes the results of the

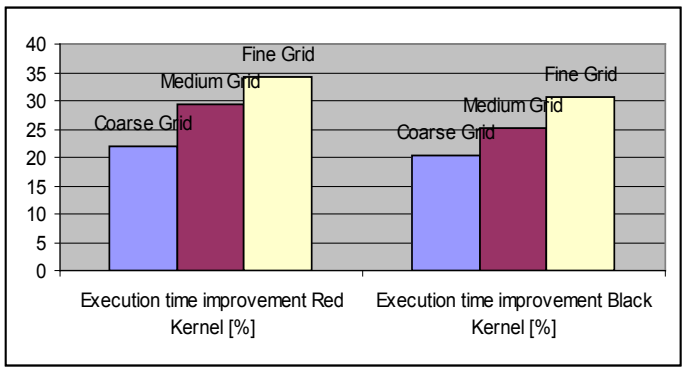

Figure 6. Execution time improvement for various kernels

TABLE III. SPEED-UP VALUES FOR THREE DIFFERENT GRAINED GRIDS

Iterations	Coarse Grid	Medium Grid	Fine Grid
100	0.396	2.03	5.36
300	0.852	3.93	11.21
500	1.117	4.89	11.68
1000	1.454	6.17	12.11
3000	1.827	7.51	12.49
5000	1.927	7.86	12.56
10000	2.009	8.17	12.64

comparison between the three tests. The numbers show that the GPU algorithm performs best on the finest grid, because of its tremendous parallel power, which is not fully exploited on coarse grids.

Finally fig. 7 presents the number of Mega cells computed per second for each of the grids. One can see that the values seem to asymptotically reach a steady state. This is caused by the fact that the memory copies which are performed at the beginning and at the end of the algorithm decrease in weight as more iterations are executed. The figure also shows that the switch from the coarsest grid to the medium leads to a greater performance improvement than the switch from the medium to the finest one. This is caused by the reduced number of blocks which are executed for the coarsest grid, and thus by the poor initial occupancy.

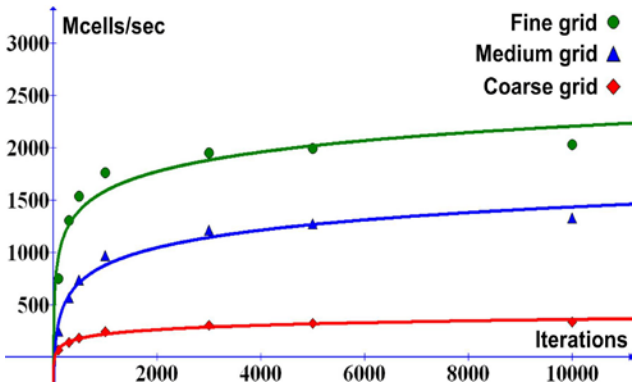

Figure 7. Performance comparison of the GPU version for the three grids

VI. CONCLUSIONS

The research activity described in this paper has been focused on the optimization of stencil based GPU kernels. Four different kernel versions have been developed until the optimum solution has been found. Section four has shown that the optimization of GPU code is not a simple task because of the complex dependencies between the various parameters.

Although the most demanding parts of the algorithm have been moved to the GPU, the CPU still plays an important role, namely to allocate and initialize the buffers, to call the two kernels with swapped input parameters at every iteration, and to copy the final results back to the CPU memory.

Section five has shown that in contrast to previous approaches [16-17], which use either the "Coalesced" or the "Full shared memory support" version, the "Partial shared memory support" version represents an improvement of 11%, respectively 27.33% for the red kernel and of 5.9%, respectively 24.52% for the black kernel. The approach described in the paper does not only represent a step forward for the steady state heat conduction problem but also for any other algorithm which performs the numerical solution of partial differential equations or which is stencil based.

We have also compared the performances of the CPU and GPU algorithms on three different grained grids (section five has presented a detailed description of the comparison). The speed-up factor varies from just over two to well over one order of magnitude for the finest grid (the reason for this great variation is the reduced number of thread blocks which are executed on the coarser grids).

ACKNOWLEDGMENT

This paper is supported by the Sectoral Operational Programme Human Resources Development (SOP HRD), financed from the European Social Fund and by the Romanian Government under the contract number POSDRU/88/1.5/S/76945.

REFERENCES

[1] D. Kirk, W.M. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, London, 2010.

[2] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, "GPU Computing", Proc. of the IEEE, Vol. 96, pp. 879-884, 2008.

[3] C. Zou, C. Xia, G. Zhao, "Numerical Parallel Processing Based on GPU with CUDA Architecture", Inter. Conf. on Wireless Networks and Information Systems, pp. 93-96, 2009.

[4] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, S.Z. Ueng, S.S. Baghsorkhi, "Program Optimization Carving for GPU Computing", Journal of Parallel and Distributed Computing, Vol. 68, No.10, pp. 1389-1401, Oct. 2008.

[5] M. Blazewicz, K. Kurowski, B. Ludwiczak, K. Napierala, "Problems Related to Parallelization of CFD Algorithms on GPU, Multi-GPU and Hybrid Architectures", AIP Conference Proceedings, Rhodes, Greece, vol.1281, pp. 1301-1304, 2009.

[6] N. Takada, T. Shimobaba, N. Masuda, and T. Ito, "High-Speed FDTD Simulation Algorithm for GPU with Compute Unified Device Architecture", IEEE Antennas and Propagation Society International Symposium, Charleston, USA, pp. 1-4, July 2009,.

[7] J.F. Wendt, Computational Fluid Dynamics: An Introduction, 3rd Edition, Springer, Berlin, Germania, 2009.

[8] T.J. Chung, Computational Fluid Dynamics, Cambridge University Press, Cambridge, UK, 2002.

[9] J. Blazek, Computational Fluid Dynamics: Principles and Applications" 2nd Edition, Elsevier, London, UK, 2007 .

[10] K.A. Hoffmann, S.T., Chiang, Computational Fluid Dynamics, Vol. 1, Engineering Education System, Wichita, USA, 1998.

[11] Q. Jin, D.B. Thomas, W. Luk, "Exploring Reconfigurable Architectures For Explicit Finite Difference Option Pricing Models", Inter. Conf. on Field Programmable Logic and Applications, Prague, Czech Rep., pp. 73-78, Aug. 2009.

[12] A.M. Bruaset, A. Tveito, Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, New York, USA, 2006.

[13] G. Chen, G. Li, S. Pei, Baifeng Wu, "High Performance Computing Via a GPU", 1st Inter. Conf. on Information Science and Engineering 238-241, 2009.

[14] NVIDIA Corporation. CUDA, Compute Unified Device Architecture Programming guide v3.1, 2010.

[15] NVIDIA Corporation. CUDA, Compute Unified Device Architecture Best practices guide v3.1, 2010.

[16] J. Cohen, "Solving PDEs on Regular Grids with OpenCurrent", GPU Technology Conference, San Jose, USA, Sept. 2010.

[17] D. Donno, A. Esposito, L. Tarricone, L., Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD", IEEE Antenas and Propagation Magazine, Vol. 52(3), pp. 116-122, June 2010.