Multigrid Method for Linear Complementarity Problem and Its Implementation on GPU

Vladimír Klement, Tomáš Oberhuber

Abstract—**We present the CUDA implementation of the parallel multigrid solver for the linear complementarity problem. As a smoother, the Projected SOR method is used. We describe implementation of all parts of the multigrid cycle on GPU. We explain, what specific properties of the GPU must be taken into account during the parallelization. The efficiency of the final algorithm is demonstrated on the constrained level-set method used in image segmentation. For this task, the speed-up up to 3 was achieved on Nvidia GeForce GTX 480 compared to more expensive 12 core AMD Opteron.**

Index Terms—**complementarity problem, geometric multigrid, projected SOR, GPU.**

I. Introduction

We consider the linear complementarity problem in the following form

$$\mathbb{A}\mathbf{x} \geq \mathbf{b}, \quad (1)$$

$$\mathbf{x} \geq \mathbf{c}, \quad (2)$$

$$(\mathbb{A}\mathbf{x} - \mathbf{b})^T (\mathbf{x} - \mathbf{c}) = \mathbf{0}, \quad (3)$$

where $\mathbb{A}$ is symmetric positive definite matrix and $\mathbf{c}$ is a given vector (constraint). This kind of problems arises in computational mechanics [5], financial mathematics [8] and/or image processing [6] and can be solved by the Projected SOR (PSOR) method introduced in [10]. If such problem arises from the discretization of a PDE with a constraint, the geometric multigrid method [3] with the PSOR method used as a smoother can be applied [8], [14]. We present a parallelization of this method which is applicable even to problems with sparse matrix $\mathbb{A}$. The parallel solver is implemented in CUDA to run on the GPU.

II. Contribution

We have developed a parallel algorithm for geometric multigrid method with the PSOR method as smoother which runs efficiently on GPUs. The algorithm is based on matrix coloring and works well with sparse matrices. To show its efficiency we solve constrained level-set method in image segmentation. Even though the test matrix is symmetric and penta-diagonal these properties are only used during creation of transition operators and coarser system matrices, main iterative part of the algorithm (multigrid cycle) can work with general matrices. Therefore in future it can be used in combination with Algebraic multigrid, which is able to automatically create transition operators and coarser system matrices just from the original system matrix, to solve larger range or problems.

Manuscript received January 1, 2014; revised July 26, 2014, August 7, 2014, August 20, 2014, September 6, 2014 and September 17,2014.

V. Klement and T. Oberhuber PhD. are with the Department of Math, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, Břehová 7, Prague, Czech Republic.

E-mails: vladimir.klement@fjfi.cvut.cz, tomas.oberhuber@fjfi.cvut.cz.

III. Projected SOR method

The PSOR method [10] is a modification of the well known SOR method. We repeat the following updating of the vector $\tilde{\mathbf{x}}^{(p)}$:

$$\mathbf{r}_i^p = \left(\mathbf{b}_i - \sum_{j<i} a_{ij}\tilde{\mathbf{x}}_j^{(p+1)} - \sum_{j>i} a_{ij}\tilde{\mathbf{x}}_j^{(p)} \right),$$

$$\hat{\mathbf{x}}_i^{(p+1)} = (1-\omega)\,\tilde{\mathbf{x}}_i^{(p)} + \frac{\omega}{a_{ii}} r_i^p, \quad (4)$$

$$\tilde{\mathbf{x}}_i^{(p+1)} = \max\{\hat{\mathbf{x}}_i^{(p+1)}, \mathbf{c}_i\}, \quad (5)$$

until the prescribed tolerance level for the modified (zero for restricted points, i.e. points where $x_i = c_i$) ℓ^2 residual norm is achieved. The equation (4) is, in fact, common SOR method and the equation (5) ensures that the prescribed constraint $\mathbf{x} \geq \mathbf{c}$ is satisfied. For the convergence study of this method we refer to [10]. However, to improve overall convergence, we use the multigrid method for solving the problem instead of the plain PSOR and PSOR is used as a smoother in it. In the following section, we briefly summarize essential parts of the geometric multigrid method.

IV. Geometric multigrid

Term multigrid methods [3], [7] covers a group of methods for solving linear systems arising from the discretization of PDEs. The multigrid methods profit from use of hierarchical structure of grids with different numbers of elements.

Conventional stationary iterative methods (like Jacobi, Gauss-Seidel or SOR) quickly eliminate the oscillatory components of the error (high frequencies), while the smooth components have low reduction rate of $1 - O(h^2)$. This renders them inefficient for large linear systems. To overcome this issue multigrid methods solve problem simultaneously on coarser grids. The former smooth components become more oscillatory on them and thus can be easily eliminated by the stationary solvers (referred to as *smoothers* in the context of multigrid algorithms). This makes the multigrid methods much faster than standard solvers.

To formulate problem on coarser grids and use solution from them to improve solution on the finer grids, the so called *transition operators* are needed. The exact appearance of these operators depends on the chosen discretization, but they can be computed before the main iterative cycle and stored as matrices. Their application then consist only in one matrix vector multiplication.

Projected multigrid method [8] modifies the standard multigrid cycle to be able to solve linear complementarity problems. This is done by transferring also the solution constraints on the coarser grids. Projected multigrid *V-cycle* then looks as follows:

Algorithm 1: Projected multigrid *V-cycle*

```
1) Start with initial approximate solution $x_0^h$
2) Relax (do few smoother iterations) the current solution to get new estimate $x^h$
3) Compute the fine-grid residual $r^h = b^h - A^h x^h$
4) Compute difference between given constraint and current solution $d^h = c^h - x^h$
5) Restrict residual and constraint difference to the coarse grid by restriction operator $r^{2h} = I_h^{2h} r^h, d^{2h} = I_h^{2h} d^h$
6) Solve $A^{2h} e^{2h} = r^{2h}$ with constraint $d^{2h}$
7) Interpolate error correction to the fine grid by prolongation operator $e^h = I_{2h}^h e^{2h}$
8) Correct current solution $x^h = x^h + e^h$
9) Repeat from 2 (if needed)
```

This is a case of two grid hierarchy. If more grids are to be used, one simply replaces the direct solution of the coarse-grid problem with a recursive call to this algorithm on all grids except the coarsest one.

We use PSOR method both as a smoother for finer grids and as a solver on the coarsest one.

V. Parallel algorithm

The parallelization of the algorithm mentioned above consists of the following steps: 1) Parallelization of the coarse grid solver, 2) parallelization of the transition operators, 3) parallelization of residual calculation and 4) parallelization of the underlying smoother.

As a coarsest grid solver, some direct solver is often used. However its parallelization can be difficult, so we rather used the fine grid smoother also for this task. This shouldn't hinder the final performance, because not much time is spent on the coarsest grid.

Parallelization of the transition operators and residual calculation is rather trivial. Both these tasks are basically a simple matrix vector multiplication combined with vector subtraction (residual) or addition (improvement of fine grid solution). The parallelization of the smoother is discussed in the following text,

Essential issue with SOR parallelization is that it is (same as Gauss-Seidel method) inherently sequential. To compute x_i^{n+1} we need to know x_{i-1}^{n+1}. A remedy is to decompose the matrix $\mathbb{A}$ into several independent components (or colors) and process them separately. This method is known as red-black [11] or in general multicolor/vertex [1], [2], [4] coloring.

Such decomposition must meet following conditions: Let $P(i)$ denote the set of the column indices of the non-zero elements of the i-th row of matrix $\mathbb{A}$ minus index i ,i.e., $P(i) \equiv \{k \in 1, \dots, N \mid k \neq i \wedge a_{ik} \neq 0\}$. Then the coloring of the set of all indices $1, \dots, N$ is the system of disjoint subsets $\mathcal{N} = \{N_1, \dots, N_m\}$ such that $N_1 \cup N_2 \cup \dots \cup N_m \equiv \{1, \dots, N\}$ and

$$(\forall j \in \{1, \dots, m\})(\forall i \in N_j)(P(i) \cap N_j = 0). \qquad (6)$$

Let us denote $\mathcal{N}_k = \bigcup_{i=1}^{k} N_i$ and $\mathcal{N}_0 = \emptyset$. Then for any color $k \in \{1, \dots, m\}$ the recurrences

$$\begin{aligned} r_i^n &= \left(b_i - \sum_{j \in \mathcal{N}_{k-1}} a_{ij} x_j^{n+1} - \sum_{j \notin \mathcal{N}_{k-1} \cup \{i\}} a_{ij} x_j^n \right), \\ x_i^{n+1} &= \frac{\omega}{a_{ii}} r_i^n + (1 - \omega) x_i^n \quad \text{for all } i \in N_k, \end{aligned} \qquad (7)$$

are independent (in the meaning that we can process them in any order) and therefore they can be processed in parallel. Individual colors, however, must still be processed sequentially. Consider the following algorithm:

Algorithm 2: Multi-colored SOR

```
1) Decompose the set $\{1, \dots, N\}$ into a system of disjoint subsets $\mathcal{N} = \{N_1, \dots, N_m\}$ such that (6) holds, set $n = 0$ and $x_i^0 = x_i^{ini}$ for all $i = 1, \dots, N$ and some initial state $x^{ini}$.
2) Repeat until convergence:
3)     for $k = 1, \dots, m$ do (sequentially)
4)         For all $i \in N_k$ update $x_i^n$ in parallel by (7).
```

In our case we can use simple red-black coloring, since our problems are defined on uniform square grid.

The unknowns are also reordered by this coloring, so that elements of the same color are grouped together, taking the red ones are first. For example the system matrix is changed so that its first half contains only rows belonging to the red elements and the second half only the rows belonging to the black ones (see fig. 1 for visualization).

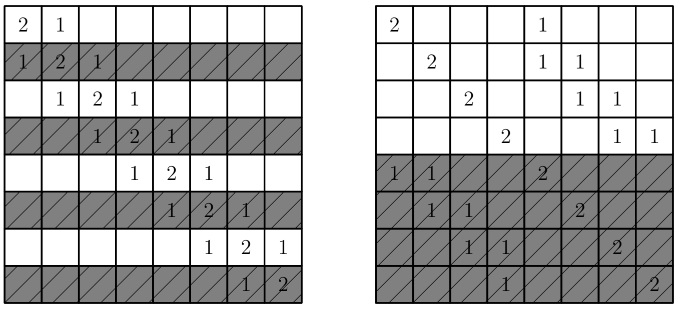
Fig. 1. Red-black reordering of the sample 1D system matrix. On the left is matrix using natural element ordering and on the right matrix using red-black ordering.

Generally it means that the system of $\mathbb{A}\mathbf{x} = \mathbf{b}$ becomes

$$\begin{bmatrix} D_r & U_{rb} \\ L_{rb} & D_b \end{bmatrix} \begin{bmatrix} x_r \\ x_b \end{bmatrix} = \begin{bmatrix} b_r \\ b_b \end{bmatrix}, \qquad (8)$$

where D_r and D_b are diagonal matrices, so from this form it can be clearly seen that update of red elements only depend on the values of black elements and vice versa.

This reordering is done because PSOR method in each phase processes only elements of one color and so it needs to read only those matrix rows, that belong to this elements. Therefore it is advantageous to have them in continuous block of memory. It also simplifies processing of the elements of only one color, because it is not needed to compute or fetch the color of each element and then skip elements with the wrong one, instead based on the currently processed color only the first or the second half of the elements is updated.

Of course all vectors and transitions operators must be changed accordingly, but this is done already during the creation of all system matrices, vectors and transition operators, so it imposes no additional work or change of the used algorithms (all multigrid parts only expect ordering to be consistent, but does not require any particular one).

VI. GPU Architecture

GPUs are highly parallel devices with shared memory similar to vector architectures. They have evolved from simple graphics accelerators to powerful tools for the high performance computing. Unlike the classical multi-core processors which have at most tens of cores, they consist of hundreds to thousands of computational units. Therefore they are best utilized on massively parallel problems.

Compared to other types of parallel programming (i.e. OpenMP, MPI), programming for GPU has some specifics given by the type of calculations the GPUs were designed for. It is important to know them and keep them in mind when creating programs for GPU in order to fully utilize its potential. In the rest of this section the most important ones will be pointed out:

- **Dedicated memory** GPU does not use standard RAM, instead it has its own video RAM referred as the global memory. This isn't issue when problem is completely solved on GPU, but in the case of converting only the most computationally demanding parts on the GPU and doing rest of the work on the processor, constant copying can easily become a bottleneck.
- **Branching** The multiprocessors can be compared to SIMD architecture. Each thread must process the same instructions as the others. Therefore each 32 consecutive threads are synchronized implicitly and they are called the *warp*. Because of this, branching of threads in kernels can lead to inefficiencies. If the condition result isn't same for all threads in the warp, both branches must be taken by all warp threads, even though each of them will only work in their correct branch and idly wait in the other. This situation is called *threads divergence* or that *threads diverge* and can be major issue for some types of the problems see [15].
- **Coalescing** Graphics card have much bigger bandwidth than standard RAM when reading blocks of data. More precisely when one warp try to read or write continuous block of data it can be coalesced into single operation and so whole block can be loaded more than thirty times faster. This is extremely important feature for bandwidth limited problems
- **Thread hierarchy** Computational threads form a two layer hierarchy. On the first one, the threads are grouped into blocks. On the second one, all blocks create the so called grid. Number of blocks in the grid is completely up to the programmer and it should match the size of the solved problem. Size of the block can be also chosen, however it must be at most 1024. The reason for this two level hierarchy is that only threads that are in the same block can communicate between each other. This means that blocks have to be completely independent.

VII. Multigrid GPU parallelization

The parallel code for the GPU is written in CUDA framework [13]. The algorithms described above are used

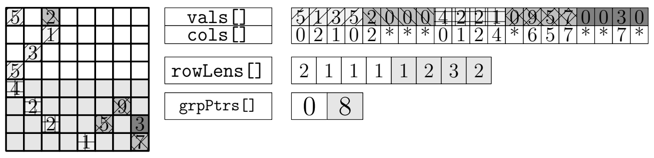

Fig. 2. Visuazlization of RgCSR format. For simplicity each group contains only 4 rows instead of 32. General matrix is used as an example to better show all properties of RgCSR. Image taken from [12]

without any modification. The only difference is that special sparse matrix format was used on GPU in order to achieve coalesced accesses (CPU version uses standard CSR).

All matrices are therefore stored in our own RgCSR format for general sparse matrices described in [12]. It works (fig. 2) so that it groups 32 matrix rows together. Each such group is transposed and stored in the Ellpack format. This means that the non-zero matrix element are not ordered by rows but by their appearance in the rows. More exactly firstly the first non-zero elements in the rows are stored into a sequential block. Then they are followed by the block of the second non-zero elements in the rows and so on. This way we can (unlike for CSR format) achieve coalesced accesses on GPU during both sparse matrix vector multiplication and Red-black PSOR updates, note that there is no need to change algorithm 2, sparse matrix format only affects how the $\sum_{j\in\mathcal{N}_{k-1}} a_{ij}x_j^{n+1}$ and $\sum_{j\notin\mathcal{N}_{k-1}\cup\{i\}} a_{ij}x_j^{n}$ will be computed. Let us demonstrate the coalesced accesses on SpMV kernel parallelized by matrix rows (code fragment taken from [12]):

```
__global__ void SpMV_RgCSR(
  int mSize, double* vals, int* cols,
  int* grpPtrs, int* rowLens,
  double* x, double* Ax )
{
  int row = blockIdx.x *  blockDim.x
    + threadIdx.x;
  if( row >= mSize ) return;
  int grpOffset = grpPtrs[blockIdx.x];
  int currentGrpSize = blockDim.x;

  // The last group may be smaller
  if((blockIdx.x+1)*blockDim.x > mSize)
    currentGrpSize = mSize % blockDim.x;

  double product = 0.0;
  const int rowLen = rowLens[row];
  int i = grpOffset + threadIdx.x;
  for( int j = 0; j < rowLen; j++ )
  {
    //Coalesced read on vals and cols
    product += vals[i] * x[ cols[i] ];
    i += currentGrpSize;
  }

  //Coalesced write
  Ax[row] = product;
}
```

Compared to the Ellpack format, our format needs less memory when storing general matrices with different number of non-zeroes in each row, which however is not the case for our test matrices because they are all penta-diagonal. This penta-diagonality also means that special formats for diagonal matrices could have been used, but our goal was to keep most of the program able to work with general matrices and thus usable for larger range of problems.

With this changes in place it is easy to effectively convert remaining parts. Transition operators and residual calculation are SpMV multiplication whose parallelization was already described. Vector addition and subtraction (for improving the solution and residual calculation) are trivial to parrallelize and reduction operation for computing the residual norm uses algorithm from CUDA reduction example [13].

Main and most complex is the PSOR update kernel, which is however quite similar to SpMV one:

```
__global__ void PSOR_RgCSR(
int mSize, double* vals, int* cols,
int* grpPtrs, int* rowLens,
double* x, double *b, double *c
double omega, int startI, int endI)
{
  int row = blockIdx.x * blockDim.x
    + threadIdx.x + startI;
  if( row >= endI ) return;
  int grpOffset = grpPtrs[blockIdx.x];
  int currentGrpSize = blockDim.x;
  if((blockIdx.x+1)*blockDim.x > mSize)
    currentGrpSize = mSize % blockDim.x;
  const int rowLen = rowLens[row];
  int i = grpOffset + threadIdx.x;

  double diag=1, nondiag=0, xr=0;
  for( int j = 0; j < rowLen; j++ )
  {
    int col = cols[i];
    double val = vals[i];
    double xc = x[col];
    if (col==row) {diag=val; xr=xc;}
    else nonDiag += val * xc;
    i += currentGrpSize;
  }
  xr = omega*(b[row]-nonDiag)/diag
     + (1.0-omega)*xr;
  double con = c[row];
  if (xr > con) xr = con;
  x[row] = xr;
}
```

Most of the read/write operations are coalesced as well. Newly there are two branching operations that should be discussed. The first is

```
if (col==row) {diag=val; xr=xc;}
else nonDiag += val * xc;
```

which can hinder the performance if diagonal elements have different index j in each row and thus threads divergence occur. Note however that the read operation from global memory isn't part of the condition statement, so this effect is quite limited. We have tested effect of eliminating the branching issues by changing row position of diagonal elements (by moving their respective values in vals and cols arrays) and find out that

- For the worst case when diagonal elements are randomly positioned and therefore have different indeces j, there was slowdown about 3% compared to original test matrices.
- In the best case, where all diagonal elements have same index j and so no thread divergence occurs, the same performance as for the test matrices has been observed.
- Case with no branching (achieved by moving diagonal elements to the beginning of each row, so that all of them have same known index $j = 0$, which means that they can be fetched separately before the for loop and the loop processes only non-diagonal elements) was about 2% faster. This was probably due to some compiler optimization previously blocked by the condition. However this improvement wasn't implemented because it was not worth making RgCSR format more complex.

Neither the second condition

```
if (xr > con) xr = con;
```

which keeps solution bellow the given constraint should affect the performance much. Even if the threads diverge, they have to wait only for one assignment operation between variables in local registers. It is very fast and therefore quite negligible compared to the total kernel computation time.

Final note can be made about the data transfers between main CPU memory and the GPU global memory. Since our algorithm runs completely on GPU all data must be uploaded to the global memory only once and then only final results are fetched after the computation.

VIII. Experimental results

We demonstrate the efficiency of the CUDA implementation of the Projected SOR method on the constrained level-set method in image segmentation [6]. We only briefly expose the problem we solve. We assume having an image intensity function I_0 defined on the domain $\Omega \equiv (0,1)^2$. We set initial curve Γ_0 as an initial guess of the segmented object on the image. We also set two disjoint subsets Ω_{in} and Ω_{out} of the domain Ω such that Ω_{in} lies in the segmented object interior and Ω_{out} in exterior. We define function $v \in C(\Omega)$ (constraint) such that v is negative everywhere in Ω_{in} and positive everywhere in $\Omega \setminus \Omega_{in}$. Finally let $u_{ini} \in C(\Omega)$ be a level-set function such that its zero level-set equals Γ_0. We construct a time-dependent level-set function $u = u(x,t)$ such that it satisfies the following problem.

$$\begin{aligned} u_t &\leq Q\nabla \cdot \left(g^0 \frac{\nabla u}{Q} \right) \\ u(x,t) &\leq v(x) \\ \text{for} \quad & (x,t) \in \Omega \times (0,T], \end{aligned} \tag{9}$$

$$\partial_\nu u = 0 \text{ at } \partial\Omega, \tag{10}$$

$$u\,|_{t=0} = u_{ini} \text{ in } \Omega, \tag{11}$$

where Q is regularized norm of gradient of u defined by $Q = \sqrt{\epsilon^2 + |\nabla u|^2}$ and g^0 is edge detector (i.e. function going to 0 near the edges in original image and to 1 near parts of the image with constant intensity), in our case it is defined by $g^0 = 1/(1 + K|\nabla I|^2)$, where I is the original

TABLE I
PERFORMANCE COMPARISON OF THE SOLVER RUNNING ON SINGLE CORE (CORE2 DUO) AND ON THE GPU (GTX 480) FOR THREE DIFFERENT RESOLUTIONS OF THE INPUT IMAGE.

Resolution	CPU	GTX 480	
	Time	Time	Speed-up
256x256	25.3 s	2.6 s	9.7
512x512	368 s	25 s	14.7
1024x1024	5431 s	267 s	20.3

TABLE II
COMPUTATIONS TIMES OF OPENMP VERSION ON MULTI CORE OPTERON AND SPEED-UP OF GPU VERSION ON GTX 480 COMPARED TO THEM.

Cores	256x256		512x512		1024x1024	
	Time	Speed-up	Time	Speed-up	Time	Speed-up
1	50 s	19.2	669 s	26.8	10830 s	40.6
2	25.3 s	9.7	341 s	13.6	5575 s	20.9
4	14.1 s	5.4	174 s	7.0	2748 s	10.3
8	7.6 s	2.9	89 s	3.6	1392 s	5.2
12	6.4 s	2.5	61 s	2.4	935 s	3.5

grayscale image. ϵ and K are constants which were chosen by hand to $\epsilon = 1, K = 10$.

With the numerical discretisation from [6], the problem above can be converted to the following linear complementarity problem:

$$\begin{aligned} (\mathbb{A}\tilde{\mathbf{u}})_i &\leq \mathbf{b}_i \\ \tilde{\mathbf{u}}_i &\leq v_i \\ for \quad & i \in I \end{aligned} \tag{12}$$

where I is the set of all pixels of the segmented image and the matrix $\mathbb{A}$ is pentadiagonal, symmetric, diagonally dominant M-matrix (see [11]) and therefore positive definite. The matrix coloring is obtained by checker board style red-black coloring of the image pixels.

Testing system was equipped by Intel Core2 Duo 2.6 GHz and Nvidia GeForce GTX 480. OpenMP version was tested on AMD Opteron with 12 cores running on 2.2 GHz. Three different resolutions of the input image were used.

Comparison of GPU and single core version is in Table I, Table II then shows results compared to OpenMP version.

Results show that the GPU can perform the computations much faster than the standard CPU. It can outperform even more expansive multi-core system, even though the OpenMP version has very good scaling.

IX. CONCLUSION

This article presented an implementation of parallel multigrid solver for the Projected SOR method on the GPU. It shows that the GPUs are very suitable platform for such tasks and that they can be used to extensively speed-up the computation.

ACKNOWLEDGMENTS

This work was partially supported by the project Advanced Supercomputing Methods for Implementation of Mathematical Models, project of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14, 2011-13 and project No. TA01020871 of the Technological Agency of the Czech Republic, 2011-2013.

REFERENCES

[1] L. M. Adams, H. F. Jordant, “Is SOR color-blind?”, in *SIAM J. Sci. Stat. Comput.*,voL. 7, pp. 490–506, 1986.
[2] C. Berge, “Graphs and Hypergraphs”, North-Holland Publishing Company, 1973.
[3] W. L. Briggs, V. E. Henson and S.F. McCormick, “A Multigrid Tutorial”, Society for Industrial and Applied Mathematics, 2000.
[4] H. A. Haddadene, H. Issaadi, “Perfect Graphs and Vertex Coloring Problem”, in *IAENG International Journal of Applied Mathematics*, voL 39, no. 2, pp. 128-133, 2009.
[5] Y. Hailing, Z. Depei, “Solving Elastic Contact Problems with Friction as Linear Complementarity Problems Based on Incremental Variational Principles”, in *Computational Mechanics' 95*, Springer Berlin Heidelberg, pp. 1504-1509, 1995.
[6] V. Klement, T. Oberhuber, and D. Ševčovič, “On a constrained level-set method with application in image segmentation”, submitted for publication, arXiv:1105.1429v1.
[7] J. V. Lambers, “A Multigrid Block Krylov Subspace Spectral Method for Variable-Coefficient Elliptic PDE”, in *IAENG International Journal of Applied Mathematics*, voL 39, no. 4, pp. 236-246, 2009.
[8] C. W. Oosterlee, “On multigrid for linear complementarity problems with application to American-style options”, in *Electronic Transactions on Numerical Analysis*, voL. 15, no. 1, pp. 165-185, 2003.
[9] J. D. Owens et al., “A survey of General-Purpose Computation on Graphics Hardware”, in *Computer graphics forum*, Blackwell Publishing Ltd, voL 26, no. 1, pp. 80-113, 2007.
[10] O. L. Mangasarian, “Solution of symmetric linear complementarity problems by iterative methods”, in *Journal of Optimization Theory and Applications*, voL. 22, no. 4, pp. 465-485, 1977.
[11] K. Mikula, A. Sarti, “Parallel co-volume subjective surface method for 3D medical image segmentation”, in *Parametric and Geometric Deformable Models: An application in Biomaterials and Medical Imagery*, Springer Publishers, 2007.
[12] T. Oberhuber, A. Suzuki, and J. Vacata, “New row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA”, *Acta Technica*, voL. 56, pp. 447-466, 2011.
[13] Nvidia company, “Nvidia CUDA Programing Guide version 2.2”, Nvidia, 2009.
[14] I. Yavneh, “On Red-Black SOR Smoothing in Multigrid”, in *SIAM Journal on Scientific Computing*, voL. 17, no. 1, pp. 180-192, 1996.
[15] I. Chakroun, M. Mezmaz, N. Melab, A. Bendjoudi, “Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm”, in *Concurrency and Computation: Practice and Experience*, voL. 25, pp. 1121-1136, 2013.