Proceedings of the International Multiconference on Computer Science and Information Technology pp. 349–353
ISBN 978-83-60810-27-9
ISSN 1896-7094

Solving a Kind of BVP for ODEs on heterogeneous CPU + CUDA-enabled GPU Systems

Przemysław Stpiczyński
Institute of Theoretical and Applied Informatics
of the Polish Academy of Sciences, Gliwice, Poland
Department of Computer Science
Maria Curie-Skłodowska University, Lublin, Poland
Email: przem@hektor.umcs.lublin.pl

Joanna Potiopa
Department of Computer Science
Maria Curie-Skłodowska University
Lublin, Poland
Email: joannap@hektor.umcs.lublin.pl

***Abstract*—The aim of this paper is to show that a special kind of boundary value problem for second-order ordinary differential equations which reduces to the problem of solving a tridiagonal system of linear equations with almost Toeplitz structure can be efficiently solved on modern heterogeneous computer architectures based on CPU and GPU processors using an algorithm based on the *divide and conquer* method for solving linear recurrence systems with constant coefficients.**

I. Introduction

SEVERAL problems in scientific computing can be reduced to the following boundary value problem [8]:

$$-\frac{d^2u}{dx^2} = f(x) \quad \forall x \in [0,1], \tag{1}$$

where

$$u'(0) = 0 \text{ and } u(1) = 0. \tag{2}$$

Numerical solution to the problem (1)–(2) reduces to the problem of solving a tridiagonal system of linear equations. Simple algorithms based on Gaussian elimination achieve poor performance, since they do not fully utilize the underlying hardware, i.e. memory hierarchies, vector extensions and multiple (or many) processors (cores). The matrix of such systems have a special (almost Toeplitz) form and it clear that it should also be exploit. The problem can be solved in parallel using the *divide and conquer* approach [10] and novel data formats for dense matrices with the square blocked full column major order [2]. The performance of the algorithm can also be improved by using non-square tiles [11] which can better fit into L1 cache [1].

Graphical processing units (GPUs [5]) have recently been widely used for scientific computing due to their large number of parallel processors which can be exploit using the Compute Unified Device Architecture (CUDA) programming language [4]. GPUs offer very high performance at low costs for data-parallel computational tasks, when computations are carried out in single precision [3], [7]. Thus it is a good idea to develop algorithms for hybrid (heterogeneous) computer architectures where large parallelizable tasks are scheduled for execution on GPUs, while small non-parallelizable tasks should be run on CPUs (possibly using double precision to improve numerical properties of algorithms).

The aim of this paper is to show that the divide and conquer method for solving (1) can be efficiently implemented on such heterogeneous systems including a multicore CPU and CUDA-enabled GPU.

II. The method

We want to find an approximation of the solution to the problem (1) in the grid points

$$0 = x_1 < x_2 < \ldots < x_{n+1} = 1,$$

where $x_i = (i-1)h$, $h = 1/n$, $i = 1, \ldots, n+1$. Let $f_i = f(x_i)$ and $u_i = u(x_i)$. Using the approximation for the second derivative

$$u''(x_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}$$

and the boundary conditions

$$u''(0) \approx \frac{u(0-h) - 2u_1 + u_2}{h^2}, \quad u'(0) \approx \frac{u_2 - u(0-h)}{2h}$$

we get the following equations

$$\begin{aligned} 2(u_1 - u_2) &= h^2 f_1, \\ -u_{i-1} + 2u_i - u_{i+1} &= h^2 f_i, \ i = 2, \ldots, n-1, \\ -u_{n-1} + 2u_n &= h^2 f_n. \end{aligned} \tag{3}$$

Thus, we can rewrite (3) as the problem of solving the system of linear equations [8]:

$$A\mathbf{u} = \mathbf{d}, \tag{4}$$

where the matrix A is of the following form

$$A = \begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n},$$

and the vectors satisfy $\mathbf{u} = (u_1, \ldots, u_n)^T$, $\mathbf{d} = (d_1, \ldots, d_n)^T$ and $d_1 = \frac{1}{2}h^2 f_1$, $d_i = h^2 f_i$ for $i = 2, \ldots, n$. The matrix A

can be factorized as $A = LR$, where L and R are bidiagonal Toeplitz matrices

$$L = \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix} \tag{5}$$

and

$$R = \begin{pmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \\ & & & & 1 \end{pmatrix}. \tag{6}$$

The solution to the system (4) can be found using Gaussian elimination without pivoting. A simple algorithm for solving (4) based on such a factorization can comprise two sequential stages, namely solving the systems $L\mathbf{y} = \mathbf{d}$ (forward reduction) and $R\mathbf{u} = \mathbf{y}$ (back substitution). The first stage can be done using

$$\begin{cases} y_1 = d_1 \\ y_i = d_i + y_{i-1} \qquad \text{for } i = 2, \ldots, n. \end{cases} \tag{7}$$

Then (second stage) we find the final solution to the system (4) using

$$\begin{cases} u_n = y_n \\ u_i = y_i + u_{i+1} \qquad \text{for } i = n-1, n-2, \ldots, 1. \end{cases} \tag{8}$$

More sophisticated approach can be based on the *divide-and-conquer* algorithm for solving linear recurrence systems [9]. The main idea of the algorithm is to rewrite the considered systems as block-bidiagonal systems of linear equations [9], [12]. Without loss of generality, let us assume that there exist two positive integers r and s such that $rs \leq n$ and $s > 1$. The method can be used for finding $y_1, \ldots, y_{rs}$. To find $y_{rs+1}, \ldots, y_n$, we use (7) directly. For $j = 1, \ldots, r$, we define vectors

$$\mathbf{d}_j = \left(d_{(j-1)s+1}, \ldots, d_{js}\right)^T \in \mathbb{R}^s$$

and

$$\mathbf{y}_j = \left(y_{(j-1)s+1}, \ldots, y_{js}\right)^T \in \mathbb{R}^s.$$

Then we find all $\mathbf{y}_j$ using the following formula

$$\begin{cases} \mathbf{y}_1 = L_s^{-1}\mathbf{d}_1 \\ \mathbf{y}_j = L_s^{-1}\mathbf{d}_j + y_{(j-1)s}\mathbf{e} \qquad \text{for } j = 2, \ldots, r, \end{cases} \tag{9}$$

where $\mathbf{e} = (1, \ldots, 1)^T$ and the matrix $L_s \in \mathbb{R}^{s \times s}$ is of the same form as L given by (5). Analogously, we can perform the second stage using

$$\begin{cases} \mathbf{u}_r = R_s^{-1}\mathbf{y}_r \\ \mathbf{u}_j = R_s^{-1}\mathbf{y}_j + u_{js+1}\mathbf{e} \qquad \text{for } j = r-1, \ldots, 1, \end{cases} \tag{10}$$

where $\mathbf{u}_j = \left(u_{(j-1)s+1}, \ldots, u_{js}\right)^T \in \mathbb{R}^s$ and $R_s \in \mathbb{R}^{s \times s}$ is of the same form as R given by (6). However, it is better to find the following matrix

$$U = (\mathbf{u}_1, \ldots, \mathbf{u}_r) \in \mathbb{R}^{s \times r} \tag{11}$$

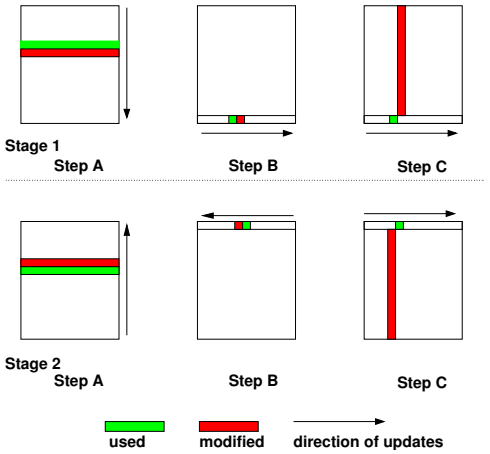

Fig. 1. *Divide and conquer* algorithm using a 2-D array

instead of individual vectors $\mathbf{u}_j$. Also, there is no need to use temporary vectors $\mathbf{y}_j$. It is clear that we have to allocate only one $s \times r$ array `U` and initially store vectors $\mathbf{d}_j$ in its columns. Then (Step 1A) we use a previously updated row to update the next row of the array. During Step 1B we update the bottom row of the table. Finally (Step 1C) we update $s-1$ first entries of each column (except for the first one). Similarly we proceed during the second stage of the algorithm (Figure 1). During Step 2A we update rows of the table *bottom-up*. Then (Step 2B) we update the first row of the table from right to left. Finally (Step 2C) using elements from the first row, we update the remaining entries of each column (except for the last one).

Both stages can be easily vectorized and parallelized. The array can be divided into blocks of columns and each processor can be responsible for computing one block. Steps A and C can be performed in parallel but steps B (in both stages) are sequential.

III. CUDA-BASED IMPLEMENTATION OF THE ALGORITHM

The detailed description of nVIDIA CUDA architecture can be found in [6]. Such a *computing device* (usually GPU) comprises a number of streaming multiprocessors (SM). Each SM consists of eight scalar cores (streaming processors, SP). SMs are responsible for executing blocks of threads. Threads in a block are grouped into so-called *warps*, each consisting of 32, that are managed and executed together. A device has its own memory system including the *global memory* (large but slow), *constant* and *texture* read-only memories providing reduction of memory latency. Each SM has also a 16 kB of fast *shared memory* that can be used for sharing data among threads within a block. The global memory access can be improved by coalesced access by all threads of a half-warp. Threads must access either 4-byte words in one 64-byte memory transaction, or 8-byte words in one 128-byte memory transaction. All 16 words must lie in the same memory segment and threads must access words in a sequence, namely k-th thread in a half-warp must access the k-th word in a segment [6]. CUDA programs consist of a number of C functions called *kernels* that are to be executed on devices as

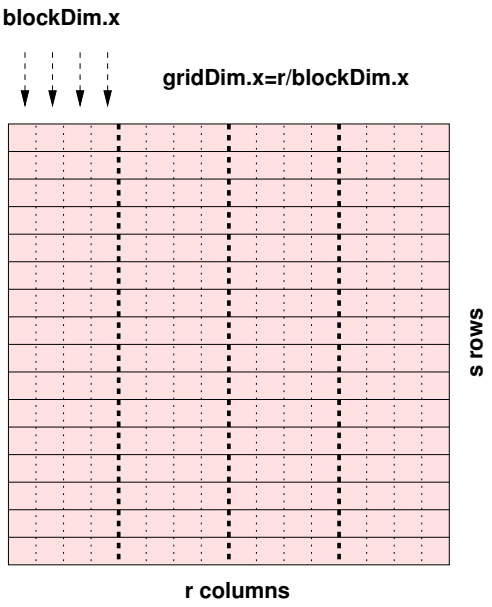

Fig. 2. Storage for the *divide and conquer* algorithm on GPUs

threads. Kernels are called from programs executed on CPUs. Host programs are also responsible for allocation of variables in the device global memory. The are also some CUDA API-functions used to copy data between host and device global memories. Each thread can identify its number, the number of its block and the block size using built-in variables `threadIdx`, `blockIdx` and `blockDim`, respectively.

Now let us consider the implementation of the *divide and conquer* algorithm. The basic idea is depicted in Figure 2. To allow coalesced memory access, elements of the $s \times r$ array U should be stored row-wise in the global memory of a device. Each thread is responsible for computing one column of the array. The number of its column can be computed as follows:

```
m = blockIdx.x*blockDim.x+threadIdx.x;
```

Each block of threads is responsible for computing one *panel* – a group of adjacent columns. For simplicity, we assume that

$$r = (\#blocks) \times (\#threads\ in\ block).$$

The source code of the host function is shown in Figure 3. Just after the array in the global memory of a device and auxiliary arrays in the host memory are allocated, the first kernel is executed to initialize the array (Figure 4). Note that the parallelized steps, namely 1A, 1C, 2A and 2C, are executed as kernels (see Figures 5–7), while sequential steps 1B and 2B are executed on CPU using double precision. In case of Step 1B, the bottom row of the array is copied from the device global memory, then it is updated using (9) and finally it is sent back to the device global memory. Analogously during Step 2B, the top row of the array is copied and updated using (10).

IV. Results of experiments

Both considered algorithms: **A1** – sequential based on (7) and (8), and **A2** – described in the previous section, have been

```
void cuda_bvp(int r, s, bsize, float *u_h){
/*
  s,r   - the number of rows and columns in the array
  bsize - the number of CUDA threads in a block
  u_h   - the array for the solution
*/

  int j,n=r*s;
  size_t size = N * sizeof(float);

// allocate arrays on host & device
  cudaMalloc((void **) &u_d, size);
  float  vf_tmp = (float *)malloc(r*sizeof(float));
  double vd_tmp = (double *)malloc(r*sizeof(double));

// initialize the s x r array on the device
  cuda_bvp_set <<< r/bsize, bsize >>> (u_d, s, r);

// perform Step 1A on the device
  cuda_bvp_1a <<< r/bsize, bsize >>> (u_d, s, r);

// perform Step 1B on CPU using double precision
  cudaMemcpy(vs_tmp, &u_d[(s-1)*r], sizeof(float)*r,
                                  cudaMemcpyDeviceToHost);
  vd_tmp[0]=vs_tmp[0];
  for(j=1;j<r;j++){
    vd_tmp[j] = (double)vs_tmp[j]+ vd_tmp[j-1];
    vs_tmp[j] = (float)vd_tmp[j];
  }
  cudaMemcpy(&u_d[(s-1)*r], vs_tmp, sizeof(float)*r,
                                  cudaMemcpyHostToDevice);

// perform step 1C and 2A on the device
  cuda_bvp_1c2a <<< r/bsize, bsize >>> (u_d, s, r);

// perform Step 2B on CPU using double precision
  cudaMemcpy(vs_tmp, u_d, sizeof(float)*r,
                                  cudaMemcpyDeviceToHost);
  vd_tmp[r-1]=vs_tmp[r-1];
  for(j=r-2;j>=0;j--){
    vd_tmp[j] = (double)vs_tmp[j]+vd_tmp[j+1];
    vs_tmp[j]=(float)vd_tmp[j];
  }
  cudaMemcpy(u_d, vs_tmp, sizeof(float)*r,
                                  cudaMemcpyHostToDevice);

// perform Step 2C on the device
  cuda_bvp_2c <<< r/bsize, bsize >>> (u_d, s, r);

// copy results to the host array
  cudaMemcpy(u_h, u_d, sizeof(float)*N,
                                  cudaMemcpyDeviceToHost);
}
```

Fig. 3. The source code of the *divide and conquer* algorithm

tested on a computer with Intel Core2 Duo (2.66 MHz, 4GB RAM) and nVidia GeForce GTX 260 (216 cores, 1792MB RAM) running under Linux with `gcc` and nVidia `nvcc` compilers for various equations of the form (1) and problem sizes (Table I, II and III, IV). To observe the accuracy of the algorithms, we have considered the following special cases:

- **A1S** – sequential algorithm based on (7) and (8) using single precision, executed on CPU,
- **A1D** – sequential algorithm based on (7) and (8) using double precision, executed on CPU,
- **A2S** – parallel algorithm, steps 1A, 1C, 2A, 2C executed on GPU, steps 1B, 2B executed on CPU, all compu-

```
__global__ void cuda_bvp_set(float *u, int s,int r{

  int k,m;
  float h=1.0/((float)(s*r));
  float h2=h*h;

// the number of my column
  m=blockIdx.x*blockDim.x+threadIdx.x;

  for(k=0;k<s;k++){
    u[m+k*r] = h2*f(h*(m*s+k));
    // f should be replaced by
    // the right hand function !!!
  }

  if((blockIdx.x==0)&&(threadIdx.x==0))
    u[0]*=0.5;
}
```

Fig. 4. CUDA kernel for the initialization of the array

```
__global__ void cuda_bvp_1a(float *u, int s, int r){

  int k,m;

  m=blockIdx.x*blockDim.x+threadIdx.x;

  for(k=1;k<s;k++)
    u[m+k*r]+=u[m+(k-1)*r];
}
```

Fig. 5. CUDA kernel for Step 1A

```
__global__ void cuda_bvp_1c2a(float *u, int s,int r){

  int m,k;

  m=blockIdx.x*blockDim.x+threadIdx.x;

// step 1C
  if(m>0) {
    float a=u[(s-1)*r+m-1];
    for (k=0;k<s-1;k++)
      u[m+k*r]+=a;
  }

// step 2A
  for(k=s-2;k>=0;k--)
    u[m+k*r]+=u[m+(k+1)*r];
}
```

Fig. 6. CUDA kernel for Step 2A and Step 1C

```
__global__ void cuda_bvp_2c(float *u, int s, int r) {
  int m,k;

  m=blockIdx.x*blockDim.x+threadIdx.x;

  if(m!=r-1) {
    float a=u[m+1];
    for (k=1;k<s;k++)
      u[m+k*r]+=a;
  }
}
```

Fig. 7. CUDA kernel for Step 2C

tations in single precision,

A2D – the same as **A2S**, but all computations in double precision,

A2SD – parallel algorithm, steps 1A, 1C, 2A, 2C executed on GPU (using single precision), steps 1B, 2B executed on CPU (double precision).

As exemplary test problems, let us consider the following.

P1: Solve

$$-\frac{d^2u}{dx^2} = \frac{\pi^2}{4}\cos(\frac{\pi}{2}x) \quad \forall x \in [0,1], \tag{12}$$

with boundary conditions (2). The exact solution to (12) is as follows

$$u(x) = \cos(\frac{\pi}{2}x).$$

P2: Solve

$$-\frac{d^2u}{dx^2} = 20000e^{-100x^2}(1-200x^2) \quad \forall x \in [0,1], \tag{13}$$

with boundary conditions (2). The exact solution to (13) is as follows

$$u(x) = 100e^{-100x^2} - 100e^{-100}.$$

Tables I and II show the accuracy of the considered algorithms for both problems. The relative error of the computed solution is calculated due to

$$error = \frac{\|\mathbf{u} - \bar{\mathbf{u}}\|_2}{\|\mathbf{u}\|_2} \tag{14}$$

where $\mathbf{u}$ and $\bar{\mathbf{u}}$ are the exact and computed solutions respectively, and $\|\cdot\|_2$ is the Euclidean norm. Tables III and IV show execution time (in seconds) of **A1D**, **A2D** and **A2SD**, and speedup of **A2D** over **A1D**. Note that it is impossible to run **A2D** on our GPU for $n = 268435456$ (2048MB of RAM required, but only 1792MB available), thus tables III and IV do not contain results for **A2D** and $n = 268435456$. Moreover, for this case, tables I and II show the results obtained on CPU using the nVidia compiler simulation feature (the option `-deviceemu`). We can observe that

- the algorithm **A1S** (single precision) achieves very poor accuracy, the use of double precision (Algorithm **A1D**) gives much better results,
- the algorithm **A2S** (single precision) achieves reasonable accuracy; the use of double precision during the steps 1B and 2B (Algorithm **A2SD**) improves the accuracy of the results,
- the algorithm **A2D** (double precision) gives better accuracy than **A1D**,
- parallel algorithms **A2D** and **A2SD** are much faster than the sequential algorithm **A1D**; the speedup grows when the problem size (value of n) grows,
- **A2SD** is about 2-3 times than **A2D** and up to 95 times faster than **A1D**, thus if someone can accept results produced by **A2SD**, the use of **A2SD** can be profitable,
- CUDA-enabled GPUs are much slower when computations are carried out using double precision, however

TABLE I
RELATIVE ERROR OF THE ALGORITHMS FOR THE PROBLEM **P1**

n	**A1S**	**A1D**	**A2S**	**A2D**	**A2SD**
1048576	1.732620e-04	1.930917e-13	6.381482e-07	1.877603e-13	2.569113e-07
4194304	3.847218e-03	2.545430e-14	9.156066e-07	1.265400e-14	2.232157e-07
16777216	2.864740e-02	5.558266e-14	3.364642e-06	1.419160e-15	3.982652e-07
67108864	6.955456e-01	1.703449e-13	2.283507e-05	2.604335e-15	2.496036e-06
268435456	9.801750e-01	1.078518e-13	1.238516e-05	4.416135e-15	3.569703e-06

TABLE II
RELATIVE ERROR OF THE ALGORITHMS FOR THE PROBLEM **P2**

n	**A1S**	**A1D**	**A2S**	**A2D**	**A2SD**
1048576	2.618970e-03	1.314334e-11	2.893085e-06	1.312754e-11	1.341402e-07
4194304	8.263400e-03	9.951346e-13	5.194502e-06	8.205207e-13	1.115430e-05
16777216	4.822094e-02	5.650246e-13	2.334787e-05	5.152762e-14	1.634096e-05
67108864	1.723210e-01	1.884351e-12	6.568387e-05	6.961652e-15	4.321754e-05
268435456	2.839243e-01	6.616377e-13	1.179028e-05	1.320262e-14	6.531590e-06

TABLE III
EXECUTION TIME (SEC.) OF THE ALGORITHMS AND SPEEDUP OF **A2D** OVER **A1D** FOR THE PROBLEM **P1**

n	**A1D**	**A2D**	**A2SD**	**Speedup**
1048576	0.0460	0.0053	0.0029	8.62
4194304	0.1868	0.0102	0.0060	17.35
16777216	0.7570	0.0341	0.0134	22.32
67108864	3.0033	0.1167	0.0420	25.69
268435456	15.5967	N/A	0.1611	N/A

TABLE IV
EXECUTION TIME (SEC.) OF THE ALGORITHMS AND SPEEDUP OF **A2D** OVER **A1D** FOR THE PROBLEM **P2**

n	**A1D**	**A2D**	**A2SD**	**Speedup**
1048576	0.0492	0.0059	0.0040	8.23
4194304	0.2010	0.0119	0.0082	16.82
16777216	0.8076	0.0382	0.0194	21.12
67108864	3.2142	0.1284	0.0578	25.03
268435456	16.6759	N/A	0.2023	N/A

the next generation GPU architecture called Fermi is pretty much faster when double precision is used (visit `http://www.nvidia.com/` for more details), thus the use of **A2D** seems to be very attractive for such new devices.

V. CONCLUSIONS AND FUTURE WORK

We have showed that the kind of boundary value problem for second-order ordinary differential equations which reduces to the problem of solving a tridiagonal system of linear equations with almost Toeplitz structure can be efficiently solved on modern heterogeneous computer architectures based on CPU and GPU processors using the *divide and conquer* algorithm for solving linear recurrence systems with constant coefficients. The algorithm achieves reasonable accuracy and excellent speedup. We have showed that the use of double precision in sequential parts of the algorithm can improve its accuracy. In the future we will consider the problem of the numerical stability of our algorithm.

REFERENCES

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[2] F. G. Gustavson, "New generalized data structures for matrices lead to a variety of high performance algorithms," *Lect. Notes Comput. Sci.*, vol. 2328, pp. 418–436, 2002.

[3] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting graphical processing units for data-parallel scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, 2009.

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, 2008.

[5] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[6] nVIDIA, *nVIDIA CUDA Programming Guide*. nVIDIA Corporation, 2009, available at http://www.nvidia.com/.

[7] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, "Program optimization carving for GPU computing," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1389–1401, 2008.

[8] L. R. Scott, T. Clark, and B. Bagheri, *Scientific Parallel Computing*. Princeton University Press, 2005.

[9] P. Stpiczyński, "Solving linear recurrence systems using level 2 and 3 BLAS routines," *Lecture Notes in Computer Science*, vol. 3019, pp. 1059–1066, 2004.

[10] P. Stpiczyński, "Solving a kind of boundary value problem for ODEs using novel data formats for dense matrices," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, M. Ganzha, M. Paprzycki, and T. Pełech-Pilichowski, Eds., vol. 3. IEEE Computer Society Press, 2008, pp. 293–296.

[11] P. Stpiczyński, "A parallel non-square tiled algorithm for solving a kind of BVP for second-order ODEs," *Lecture Notes in Computer Science*, vol. 6067, pp. 87–94, 2010.

[12] P. Stpiczyński and M. Paprzycki, "Fully vectorized solver for linear recurrence systems with constant coefficients," in *Proceedings of VECPAR 2000 – 4th International Meeting on Vector and Parallel Processing, Porto, June 2000*. Facultade de Engerharia do Universidade do Porto, 2000, pp. 541–551.