Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 509–515

ISBN 978-83-60810-51-4

GPU-accelerated WZ Factorization with the Use of the CUBLAS Library

Beata Bylina, Jarosław Bylina
Institute of Mathematics
Marie Curie-Sklodowska University
Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland
beatas@hektor.umcs.lublin.pl
jmbylina@hektor.umcs.lublin.pl

***Abstract*—We present a novel implementation of a dense, square, non-structured matrix factorization algorithm, namely the WZ factorization — with the use of graphics processors (GPUs) and CPUs to gain a high performance at a low cost. We rewrite this factorization as operations on blocks of matrices and vectors. We have implemented our block-vector algorithm on GPUs with the use of an appropriate (and ready-to-use) GPU-accelerated mathematical library, namely the CUBLAS library. We compared the performance of our algorithm with CPU implementations. In particular, our implementation on an NVIDIA Tesla C2050 GPU outperforms a CPU-based implementation. Our results show that the algorithm scales well with the size of matrices; moreover, the larger the matrix, the better the performance. We also discuss the impact of the size of the matrix and the use of ready-to-use mathematical libraries on the numerical accuracy.**

Keywords: GPU, WZ factorization, parallel computing, matrix factorization.

I. Introduction

GRAPHICS processing units (GPUs) are useful for scientific computations, particularly for general problems such as dense linear systems, the eigen decomposition and the singular value decomposition. In this paper we focus on developing a GPU-accelerated WZ factorization with the use of the CUBLAS library [8] and a further aim is a parallel, low-cost and efficient WZ-factorization.

Solution of linear systems of the form:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \qquad \text{where} \quad \mathbf{A} \in \mathbb{R}^{n \times n}, \quad \mathbf{b} \in \mathbb{R}^{n}, \qquad (1)$$

is an important problem in scientific and engineering computations. One of the methods to solve a dense linear system (1) is a factorization of the matrix $\mathbf{A}$ (that is its decomposition to factor matrices of a simpler structure) and then solving simpler linear systems. Here we deal with the first stage only, that is the factorization of $\mathbf{A}$.

The most known factorization is the LU factorization. There are some implementations of the LU factorization for CUDA architectures [1], [7]. Authors of [1] proposed to accelerate the LU factorization on a multicore node enhanced with multiple GPU accelerators.

This work was partially supported within the project N N516 479640 of the Ministry of Science and Higher Education of the Polish Republic (MNiSW) "Modele dynamiki transmisji, sterowania zatłoczeniem i jakością usług w Internecie"

In this work we study another form of factorization, namely the WZ factorization. In [2] we showed that there are matrices for which applying the incomplete WZ preconditioning gives better results than the incomplete LU factorization.

The outline of the paper is following. In Section II we give an overview of the features of GPUs and the used ready-to-use GPU-accelerated mathematical library (CUBLAS).

Section III describes the idea of the WZ factorization [4], [10] and the way the matrix $\mathbf{A}$ is factorized to a product of matrices $\mathbf{W}$ and $\mathbf{Z}$ — such a factorization exists for a nonsingular matrix (with pivoting) what it was shown in [4].

In Section IV we describe the algorithm of the WZ factorization using the vector and matrix notation [3]. We present the algorithm for matrices which can be factorized without pivoting, that is, for example, for symmetric positive definite ones and strictly diagonally dominant ones (as it was proved in [4]). In Section IV we also present the design of a new CPU/GPU hybrid kernel for performing a WZ factorization on GPU with the use of CUBLAS.

In Section V we present details of the implementation and the results of our experiments. We benchmarked our implementation on a GPU (NVIDIA Tesla C2050) and compared it with the CPU-based implementation on a multicore architecture Intel Core i7. We tested the algorithms with square matrices of sizes from 128 to 8192. We analyzed the performance of our algorithm measuring the performance time, the efficiency (both for the single and double precision), the speed-up and the relative error. We study the influence of the size of the matrix and the use of the ready-to-use mathematical library on the numerical accuracy.

Section VI is a summary of our experiments.

II. GPU programming model

Graphics processing units (GPUs) have recently been used for many applications beyond graphics, introducing the term *general-purpose computation on graphics processing units* (GPGPU), thanks to (among others) the CUDA architecture (*Compute Unified Device Architecture*) [9] prepared by NVIDIA.

Graphics processing units are manycore computing systems, being able to deal with thousands threads processing simple data. In contrast to CPUs, GPUs have a relatively higher

effective memory clock in comparison to its computational core clock. The operations performed on each data element are independent of each other, and can be efficiently computed in parallel using many processors on a GPU.

One of the major challenges in developing GPGPU algorithms is to create techniques which fully use pipelining, many cores and the high memory bandwidth. It is not an easy task to build an efficient algorithm which uses all the GPU's features. Therefore, we propose using ready-to-use libraries. The use of such libraries requires preparation of an adequate, block version of the algorithm where matrix-vector operations are used.

Among computing tools for the CUDA architecture, we can find the CUBLAS library [8]. During programming linear algebra problems, the use of CUBLAS make an efficient application creation process much easier. The CUBLAS library is an implementation of BLAS (*Basic Linear Algebra Subprograms*) on GPUs. To use the CUBLAS library, we must send the matrices and vectors into the GPU memory space, fill them with data, call the sequence of desired CUBLAS functions, and then upload the results from the GPU memory space back to the host.

III. WZ Factorization

Here we describe shortly the WZ factorization usage to solve (1). The WZ factorization is described in [4], [6]. Assume that the $\mathbf{A}$ is a nonsingular matrix. We have to find matrices $\mathbf{W}$ and $\mathbf{Z}$ that fulfil $\mathbf{WZ} = \mathbf{A}$ and the matrices $\mathbf{W}$ and $\mathbf{Z}$ consist of following columns $\mathbf{w}_i$ and rows $\mathbf{z}_i^T$ respectively:

$$\begin{aligned}
\mathbf{w}_i &= (\underbrace{0,\ldots,0,1}_{i}, w_{i+1,i},\ldots,w_{n-i,i},0,\ldots,0)^T \\
&\quad \text{for} \quad i = 1,\ldots,m, \\
\mathbf{w}_i &= (\underbrace{0,\ldots,0,1}_{i},0,\ldots,0)^T \\
&\quad \text{for} \quad i = p, q, \\
\mathbf{w}_i &= (\underbrace{0,\ldots,0}_{n-i+1}, w_{n-i+2,i},\ldots,w_{i-1,i},1,0,\ldots,0)^T \\
&\quad \text{for} \quad i = q+1,\ldots,n, \\
\mathbf{z}_i^T &= (\underbrace{0,\ldots,0}_{i-1}, z_{ii},\ldots,z_{i,n-i+1},0,\ldots,0) \\
&\quad \text{for} \quad i = 1,\ldots,p, \\
\mathbf{z}_i^T &= (\underbrace{0,\ldots,0}_{i-1}, z_{i,n-i+1},\ldots,z_{ii},0,\ldots,0) \\
&\quad \text{for} \quad i = p+1,\ldots,n,
\end{aligned}$$

where

$$\begin{aligned}
m &= \lfloor (n-1)/2 \rfloor, \\
p &= \lfloor (n+1)/2 \rfloor, \\
q &= \lceil (n+1)/2 \rceil.
\end{aligned}$$

(see also Figure 1).

After the factorization we can solve two linear systems:

$$\begin{aligned}
\mathbf{Wc} &= \mathbf{b}, \\
\mathbf{Zx} &= \mathbf{c}
\end{aligned}$$

(where $\mathbf{c}$ is an auxiliary intermediate vector) instead of one (1).

The first algorithm (denoted **WZCPU**, Figure 2) is a traditional implementation computing the WZ factorization with the use of CPU (given the matrix $\mathbf{A}$ in the array `a`, the output matrices $\mathbf{W}$ and $\mathbf{Z}$ are in the arrays `w` and `a`, respectively).

IV. Block WZ Factorization

Now we describe a matrix-vector algorithm for the WZ factorization of the matrix $\mathbf{A}$ which is originally presented in [3]. This algorithm is only recalled here for further development (Section IV-A).

This is a sequential algorithm where we grouped and ordered the scalar operations anew, into matrix-vector operations. We are showing the algorithm without pivoting, working only for matrices for which such a WZ factorization is executable.

Let us write the matrices $\mathbf{A}$, $\mathbf{W}$, $\mathbf{Z}$ as block matrices. We can get equations presented in Figure 3.

In Figure 3, $\widehat{\mathbf{W}}$ and $\widehat{\mathbf{Z}}$ are square matrices of the same structure as the matrices $\mathbf{W}$ and $\mathbf{Z}$, respectively; $\widehat{\mathbf{A}}$ is a full square matrix; $\widehat{\mathbf{W}}$, $\widehat{\mathbf{Z}}$ and $\widehat{\mathbf{A}}$ are of the size 2 less than the size of $\mathbf{W}$, $\mathbf{Z}$ and $\mathbf{A}$; vectors $\mathbf{a}_{1*}^T$, $\mathbf{a}_{n*}^T$, $\mathbf{z}_{1*}^T$, $\mathbf{z}_{n*}^T$ are row vectors; vectors $\mathbf{a}_{*1}$, $\mathbf{a}_{*n}$, $\mathbf{w}_{*1}$, $\mathbf{w}_{*n}$ are column vectors.

From the comparison of the corresponding elements in Figure 3 we get:

$$\begin{aligned}
&a_{11} = z_{11}; \quad a_{1n} = z_{1n}; \\
&a_{n1} = z_{n1}; \quad a_{nn} = z_{nn} \\
&\mathbf{a}_{1*}^T = \mathbf{z}_{1*}^T; \quad \mathbf{a}_{n*}^T = \mathbf{z}_{n*}^T \\
&\begin{cases} \mathbf{a}_{*1} = \mathbf{w}_{*1} z_{11} + \mathbf{w}_{*n} z_{n1} \\ \mathbf{a}_{*n} = \mathbf{w}_{*1} z_{1n} + \mathbf{w}_{*n} z_{nn} \end{cases} \\
&\widehat{\mathbf{A}} = \mathbf{w}_{*1}\mathbf{z}_{1*}^T + \widehat{\mathbf{W}}\widehat{\mathbf{Z}} + \mathbf{w}_{*n}\mathbf{z}_{n*}^T
\end{aligned} \tag{2}$$

From (2) we can describe our algorithm for finding $\mathbf{W}$ and $\mathbf{Z}$ as following:

1) let the first row of the matrix $\mathbf{Z}$ be the first row of the matrix $\mathbf{A}$;
2) let the last row of the matrix $\mathbf{Z}$ be the last row of the matrix $\mathbf{A}$;
3) compute the vectors $\mathbf{w}_{*1}$ and $\mathbf{w}_{*n}$ from:

$$\begin{aligned}
\mathbf{w}_{*1} &= \alpha\mathbf{a}_{*n} - \beta\mathbf{a}_{*1}, \\
\mathbf{w}_{*n} &= \gamma\mathbf{a}_{*1} - \delta\mathbf{a}_{*n},
\end{aligned}$$

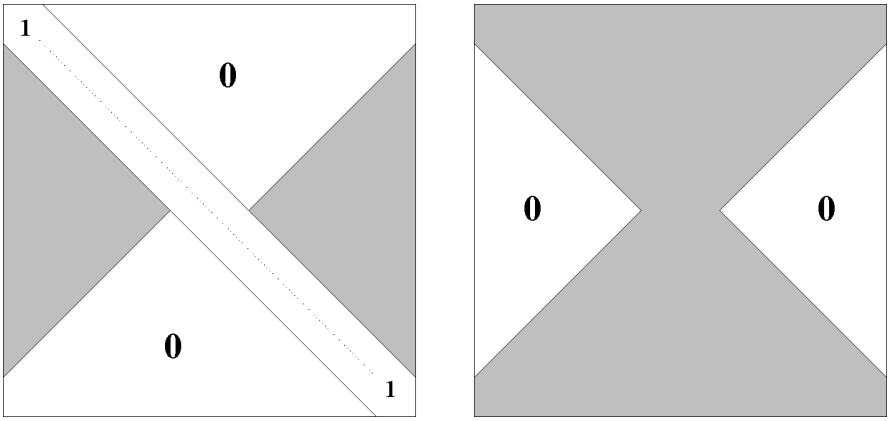

Fig. 1. Structures of the matrices $\mathbf{W}$ (left) and $\mathbf{Z}$ (right)

```
for (k=1; k<=n/2-1; k++) {
  k2 = n-k+1;
  det = a[k2][k]*a[k][k2]-a[k][k]*a[k2][k2];
  for (i=k+1; i<=(k2-1); i++) {
    w[i][k] = (a[k2][k]*a[i][k2]-a[k2][k2]*a[i][k])/det;
    w[i][k2] = (a[k][k2]*a[i][k]-a[k][k]*a[i][k2])/det;
    for (j=k+1; j<=k2-1; j++)
      a[i,j] = a[i][j]-w[i][k]*a[k][j]-w[i][k2]*a[k2][j];
  }
}
```

Fig. 2. **WZCPU**: a traditional implementation of the WZ factorization

$$\mathbf{A} = \begin{bmatrix} a_{11} & \mathbf{a}_{1*}^T & a_{1n} \\ \mathbf{a}_{*1} & \widehat{\mathbf{A}} & \mathbf{a}_{*n} \\ a_{n1} & \mathbf{a}_{n*}^T & a_{nn} \end{bmatrix} = \mathbf{WZ} = \begin{bmatrix} 1 & 0 & 0 \\ \mathbf{w}_{*1} & \widehat{\mathbf{W}} & \mathbf{w}_{*n} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_{11} & \mathbf{z}_{1*}^T & z_{1n} \\ 0 & \widehat{\mathbf{Z}} & 0 \\ z_{n1} & \mathbf{z}_{n*}^T & z_{nn} \end{bmatrix} =$$

$$= \begin{bmatrix} z_{11} & \mathbf{z}_{1*}^T & z_{1n} \\ \mathbf{w}_{*1}z_{11} + \mathbf{w}_{*n}z_{n1} & \mathbf{w}_{*1}\mathbf{z}_{1*}^T + \widehat{\mathbf{W}}\widehat{\mathbf{Z}} + \mathbf{w}_{*n}\mathbf{z}_{n*}^T & \mathbf{w}_{*1}z_{1n} + \mathbf{w}_{*n}z_{nn} \\ z_{n1} & \mathbf{z}_{n*}^T & z_{nn} \end{bmatrix}$$

Fig. 3. The WZ factorization written as blocks

where

$$\alpha = \frac{z_{n1}}{z_{1n}z_{n1} - z_{11}z_{nn}},$$

$$\beta = \frac{z_{nn}}{z_{1n}z_{n1} - z_{11}z_{nn}},$$

$$\gamma = \frac{z_{1n}}{z_{1n}z_{n1} - z_{11}z_{nn}},$$

$$\delta = \frac{z_{11}}{z_{1n}z_{n1} - z_{11}z_{nn}};$$

4) update the inner part of the matrix $\mathbf{A}$ (the matrix without its first and last row and column):

$$\widehat{\mathbf{A}}^{\text{new}} = \widehat{\mathbf{W}}\widehat{\mathbf{Z}} = \widehat{\mathbf{A}} - \mathbf{w}_{*1}\mathbf{z}_{1*}^T - \mathbf{w}_{*n}\mathbf{z}_{n*}^T;$$

5) if the size of the matrix $\widehat{\mathbf{A}}^{\text{new}}$ is 3 or more, then start over from 1., but with $\mathbf{A} = \widehat{\mathbf{A}}^{\text{new}}$, $\mathbf{W} = \widehat{\mathbf{W}}$ and $\mathbf{Z} = \widehat{\mathbf{Z}}$ (so all three matrices become smaller and smaller and the algorithm comes eventually to the end).

In the remainder of our paper we use the name **BLASCPU** to denote the next algorithm (Figure 4) — that is a matrix-vector WZ algorithm (described above) implemented on a CPU with the use of a BLAS library (input and output as in **WZCPU**).

A. GPU implementation

We show a new algorithm denoted in this paper **CUBLAS-GPU**. We have shown that matrix-vector operations play a critical role in the WZ factorization. To see how a GPU can

```
for (k=1; k<=n/2-1; k++) {
  k2 = n-k+1;
  k2_k_1 = k2-k-1;
  kplus1 = k+1;
  det = a[k2][k]*a[k][k2]-a[k][k]*a[k2][k2];
  alpha = a[k2][k]/det;
  beta = a[k2][k2]/det;
  gamma = a[k][k2]/det;
  delta = a[k][k]/det;
  cblas_dcopy(k2_k_1, &a[kplus1][k2], 1, &w[kplus1][k], 1);
  cblas_dscal(k2_k_1, alpha, &w[kplus1][k], 1);
  cblas_daxpy(k2_k_1, -beta, &a[kplus1][k], 1, &w[kplus1][k], 1);
  cblas_dcopy(k2_k_1, &a[kplus1][k], 1, &w[kplus1][k2], 1);
  cblas_dscal(k2_k_1, gamma, &w[kplus1][k2], 1);
  cblas_daxpy(k2_k_1, -delta, &a[kplus1][k2], 1, &w[kplus1][k2], 1);
  cblas_dger(CblasColMajor, k2_k_1, k2_k_1, -1, &w[kplus1][k], 1, &a[k][kplus1], n,
                                                  &a[kplus1][kplus1], n);
  cblas_dger(CblasColMajor, k2_k_1, k2_k_1, -1, &w[kplus1][k2], 1, &a[k2,kplus1], n,
                                                  &a[kplus1][kplus1], n);
}
```

Fig. 4. **BLASCPU**: a matrix-vector implementation of the WZ factorization (BLAS on a CPU)

TABLE I
THE SPECIFICATION OF THE HARDWARE AND THE SOFTWARE USED IN THE EXPERIMENTS

CPU	2 × Intel Core 3GHz (4 cores with HT)
GPU	NVIDIA Tesla C2050
OS	Debian GNU Linux 6.0
Libraries	CUDA Toolkit 3.2, MKL 10.3

be used to accelerate the WZ factorization, we implemented the algorithm on a CUDA architecture. It is the same block algorithm implemented (partially) on a GPU with the use of the CUBLAS library. The input matrix $\mathbf{A}$ should be in the GPU's memory already (in the array `a_d`) as well as there will be output matrices there (in `w` and `a_d`). It is also important that the computations are partially carried out on CPU, because we are to get four corner elements of the matrix being factorized at the beginning of each step and make some operations on them and then we use GPU (by means of CUBLAS). To copy data from CPU to the GPU and from GPU to the CPU we using `cublasSetMatrix`, `cublasGetMatrix` (of the CUBLAS library)

The algorithm **CUBLASGPU** is presented in Figure 5.

V. NUMERICAL EXPERIMENTS

The algorithms **WZCPU**, **BLASCPU** and **CUBLASGPU** were implemented with the use of the C language with the use of the single and double precision. Our codes were compiled by NVIDIA C Compiler (nvcc) with optimization flag `-O3`. Additionally, **BLASCPU** was linked with the BLAS library (*Basic Linear Algebra Subprograms*) [11] (in the MKL implementation). Similarly, **CUBLASGPU** was linked with the 3.2 CUBLAS library [8].

The algorithms were tested on an Intel Core i7 3 GHz CPU with an NVIDIA Fermi C2050 GPU working under the Linux operating system. The information about the hardware and the software used are presented in Table I. All the algorithms were run for matrices for which the WZ factorization is possible. The matrices were dense and randomly generated, of the following sizes: 128, 256, 512, 1024, 2048, 4096, 8192.

A. Time and Performance

In Figure 6 we have compared the average running time of the three WZ decomposition algorithms: implemented on a CPU without any BLAS (that is, **WZCPU**), implemented on a CPU with the use of the BLAS library in a multithreaded MKL implementation (**BLASCPU**) and implemented on a GPU with the use of the CUBLAS library (**CUBLASGPU**). They were tested both in the single and double precision.

Figure 7 presents the performance results obtained for those computations — also in the single and double precision. The performance is based on the number of floating-point operations in the WZ factorization $\left(\frac{2}{3}n^3 - \frac{1}{2}n^2 + \frac{11}{6}n - 7\right)$.

Figure 8 shows a finer analysis of **CUBLASGPU** performance, namely three main components of the running time. 'Transfer time' is the time spent by the machine on getting four values (`akk`, `ak2k`, `akk2`, `ak2k2`) from GPU to CPU (totally, in all steps of the loop). 'CPU time' is the whole time used by CPU (for computing `det`, `alpha`, `beta`, `gamma`, `delta` in all steps). Eventually, 'GPU time' is the time spent by GPU on CUBLAS functions.

Table II shows the run-time and speed-up result for the **WZCPU**, **BLASCPU** and **CUBLASGPU** algorithms running on the machine described in Table I in the double precision. In

```
for (k=1; k<=n/2-1; k++) {
  k2 = n-k+1;
  k2_k_1 = k2-k-1;
  kplus1 = k+1;
  get_from_GPU_to_CPU(&akk, &ak2k, &akk2, &ak2k2, ...);
  det = ak2k*akk2-akk*ak2k2;
  alpha = ak2k/det;
  beta = ak2k2/det;
  gamma = akk2/det;
  delta = akk/det;
  cublasDcopy(k2_k_1, &a_d[IND(kplus1,k2,n)], 1, &w[IND(kplus1,k,n)], 1);
  cublasDscal(k2_k_1, alpha, &w[IND(kplus1,k,n)], 1);
  cublasDaxpy(k2_k_1, -beta, &a_d[IND(kplus1,k,n)], 1, &w[IND(kplus1,k,n)], 1);
  cublasDcopy(k2_k_1, &a_d[IND(kplus1,k,n)], 1, &w[IND(kplus1,k2,n)], 1);
  cublasDscal(k2_k_1, gamma, &w[IND(kplus1,k2,n)], 1);
  cublasDaxpy(k2_k_1, -delta, &a_d[IND(kplus1,k2,n)], 1, &w[IND(kplus1,k2,n)], 1);
  cublasDger(k2_k_1, k2_k_1, -1, &w[IND(kplus1,k,n)], 1, &a_d[IND(k,kplus1,n)], n,
                                                        &a_d[IND(kplus1,kplus1,n)], n);
  cublasDger(k2_k_1, k2_k_1, -1, &w[IND(kplus1,k2,n)], 1, &a_d[IND(k2,kplus1,n)], n,
                                                         &a_d[IND(kplus1,kplus1,n)], n);
}
```

Fig. 5. **CUBLASGPU**: a matrix-vector implementation of the WZ factorization (CUBLAS on a GPU; here, `IND()` is a function (a macro, actually) translating two-dimensional indices into a single one-dimensional index)

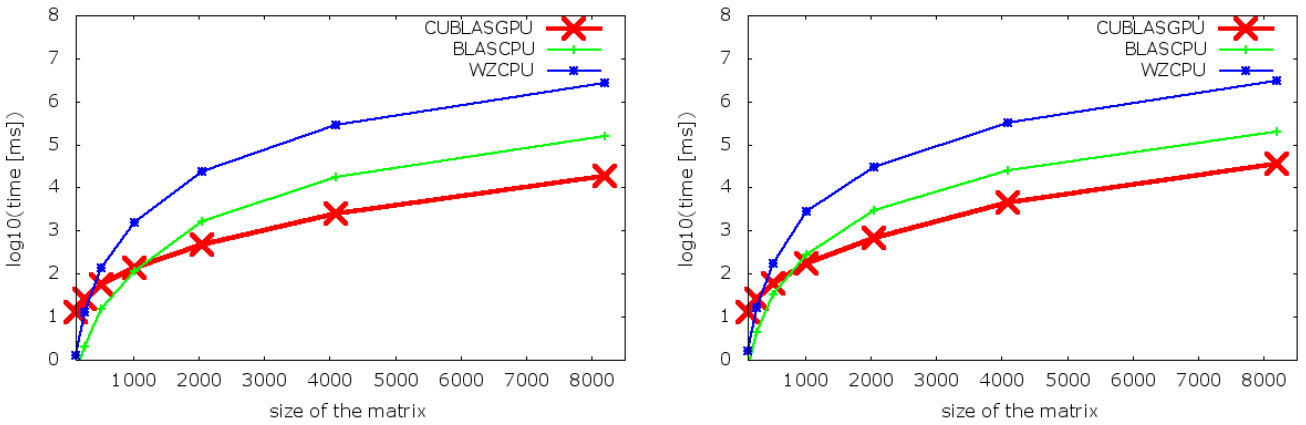

Fig. 6. The average running time of the WZ matrix decomposition as a function of the matrix size — using the single (left) and double (right) precision for the **WZCPU**, **BLASCPU** and **CUBLASGPU** algorithms (logarithmic y-axis)

Table II for the **CUBLASGPU** algorithm we give additionally (to the total runtime) the data copying time (the matrix $\mathbf{A}$) from the CPU to the GPU and the data copying time (the matrix $\mathbf{Z}$) from the GPU to the CPU. 'Total' is the sum of both the data copying time and the runtime of the **CUBLASGPU** algorithm.

Moreover, in Table II we have some relative speed-ups:

$$sp_{C:B} = \frac{\text{Time[s]}\mathbf{BLASCPU}}{\text{Time[s]}\mathbf{CUBLASGPU}}$$

and

$$sp_{C:W} = \frac{\text{Time[s]}\mathbf{WZCPU}}{\text{Time[s]}\mathbf{CUBLASGPU}}$$

(all total times).

We can see the following:

- **CUBLASGPU** is the fastest, **WZCPU** is the slowest;
- better (about 20 GFlop/s) is the performance of **CUBLASGPU** for the single precision, proportionally worse (about 10 GFlop/s) is for the double precision;
- **CUBLASGPU** scales well with the size of the matrix; moreover, bigger the matrix, better the performance;
- the time spent on the data transfer from the GPU to the CPU (in **CUBLASGPU**) is much longer than the CPU computations alone.
- the speed-ups of up to 6 times in the comparison to the multithreaded CPU solver for a matrix of a big size;

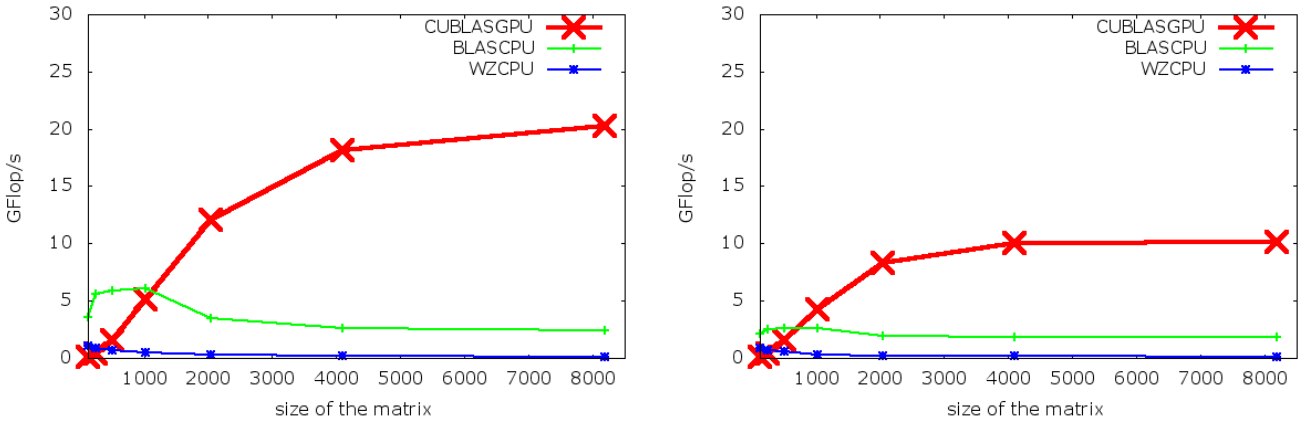

Fig. 7. The performance results for the WZ factorization — using the single (left) and double (right) precision for the **WZCPU**, **BLASCPU** and **CUBLASGPU** algorithms

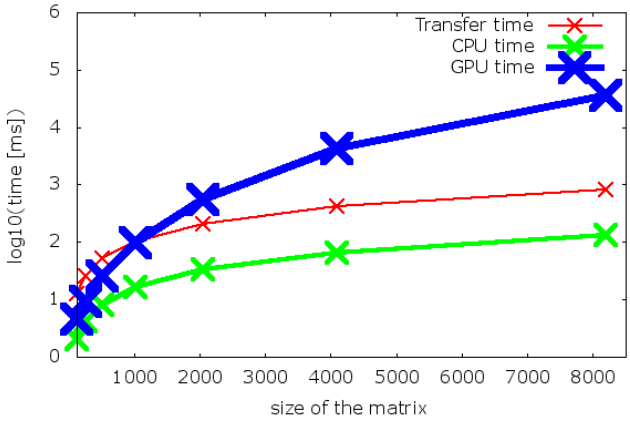

Fig. 8. The time components of **CUBLASGPU** in the double precision

TABLE II
RUN-TIMES (IN SECONDS) AND CORRESPONDING SPEED-UPS FOR THE **WZCPU**, **BLASCPU** AND **CUBLASGPU** ALGORITHMS IN THE DOUBLE PRECISION

ID	matrix size	Time[s] **WZCPU**	Time[s] **BLASCPU**	Time[s]**CUBLASGPU**				$sp_{C:W}$	$sp_{C:B}$
				CPU⟶GPU	GPU	CPU⟶GPU	Total		
1	128	1.58e+00	1.86e+00	2.23e-01	2.57e+01	2.26e-01	2.62e+01	0.06	0.07
2	256	1.75e+01	3.35e+00	4.53e-01	5.20e+01	5.04e-01	5.30e+01	0.33	0.06
3	512	1.75e+02	2.65e+01	1.19e+00	1.10e+02	1.35e+00	1.12e+02	1.56	0.24
4	1024	2.72e+03	1.10e+02	3.91e+00	2.68e+02	3.91e+00	2.75e+02	9.86	0.4
5	2048	2.98e+04	3.19e+03	1.36e+01	8.92e+02	1.42e+01	9.19e+02	32.4	3.47
6	4096	3.64e+05	2.73e+04	5.23e+01	4.95e+03	5.55e+01	5.05e+03	71.97	5.4
7	8192	3.45e+06	2.33e+05	8.41e+02	3.70e+04	2.19e+02	3.81e+04	90.65	6.12

however, for matrices of small sizes using GPU makes no sense.

B. Numerical Accuracy

The purpose of this section is not to accomplish a full study of the numerical stability and accuracy of the WZ factorization. Our goal is rather to justify experimentally that our implementation of the WZ algorithm can be used in practice.

As a measure of accuracy we took the following expression (where $||\mathbf{M}||$ is the Frobenius norm of a matrix $\mathbf{M}$) based on the relative error:

$$-\log_{10}\frac{||\mathbf{A}-\mathbf{WZ}||}{n\cdot||\mathbf{A}||}.$$

Figure 9 shows the accuracy for all the implemented algorithms (**WZCPU**, **BLASCPU** and **CUBLASGPU**) in both precisions (single and double).

From that analysis we can see that:

- the numerical accuracy decreases as the size of the matrix increases;

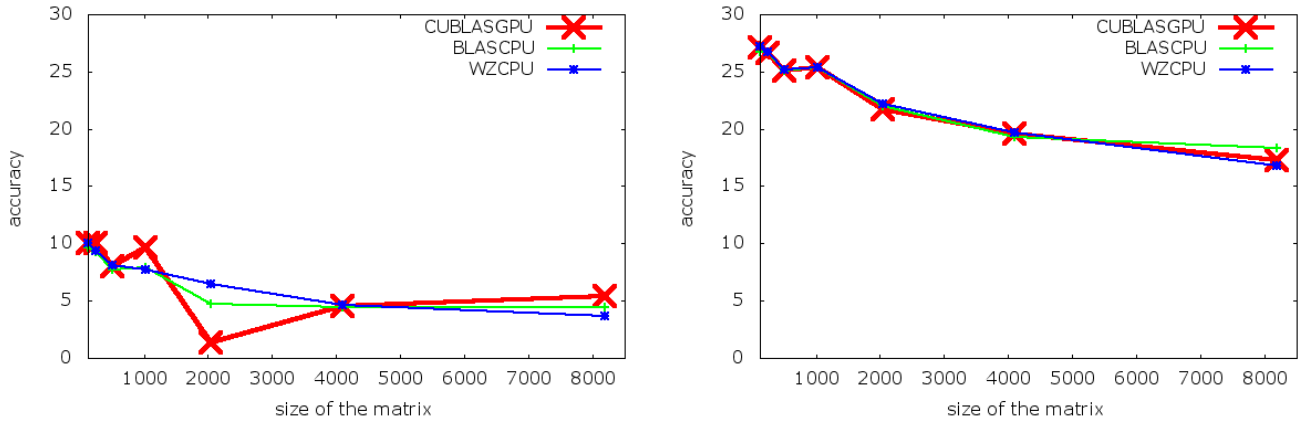

Fig. 9. The numerical accuracy $\left(-\log_{10}\frac{||\mathbf{A}-\mathbf{WZ}||}{n\cdot||\mathbf{A}||}\right)$ for random matrices $\mathbf{A}$ in the single (left) and double (right) precision for the **WZCPU**, **BLASCPU** and **CUBLASGPU** algorithms

.

- a much better accuracy for the double precision is achieved than for the single precision for all the algorithms;
- the accuracy almost does not depend on the algorithm but only on the precision (and the size of the matrix), so using the CUBLAS or BLAS libraries does not change the accuracy.

VI. Conclusion

We described an algorithm for the WZ factorization with the use of the matrix-vector operations and it was implemented with the use of the BLAS library on a CPU and the CUBLAS library on a GPU. Both the BLAS library on the CPU and the CUBLAS library on the GPU increases the speed of computations done during the WZ factorization significantly. Thanks to the libraries we were able to reach a higher performance. Speed-ups of up to 6 times for the GPU implementation in comparison to the multithreaded CPU solver are achieved for large matrices.

The major drawback of our CUBLASGPU algorithm is the data transfer from the GPU to the CPU, because it is very time-consuming and it requires to be made faster or even completely deleted. We are to improve the communication efficiency between the CPU and the GPU using the page-locked memory.

We have shown that the stability is influenced by the size of the matrix. The **CUBLASGPU** algorithm in the single precision is likely to lead to a very acceptable numerical accuracy; and its double precision implementation is much better even. The accuracy of the result is not affected when using the GPU.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, S. Tomov: LU factorization for accelerator-based systems. *9th ACS/IEEE International Conference on Computer Systems and Applications* (AICCSA 11).

[2] B. Bylina, J. Bylina: Incomplete WZ Factorization as an Alternative Method of Preconditioning for Solving Markov Chains, *Lecture Notes in Computer Science* **4967**, Springer-Verlag Berlin Heidelberg 2008, 99–107.

[3] B. Bylina, J. Bylina: The Vectorized and Parallelized Solving of Markovian Models for Optical Networks, *Lecture Notes in Computer Science* **3037**, Springer-Verlag Berlin Heidelberg 2004, 578–581.

[4] S. Chandra Sekhara Rao: Existence and uniqueness of WZ factorization, *Parallel Computing* **23** (1997) 1129–1139.

[5] D. J. Evans, M. Barulli: BSP linear solver for dense matrices, *Parallel Computing* **24** (1998), pp. 777–795.

[6] D. J. Evans, M. Hatzopoulos: The parallel solution of linear system, *Int. J. Comp. Math.* **7** (1979), pp. 227–238.

[7] R. Nath,S. Tomov, J. Dongarra: An Improved MAGMA GEMM for Fermi GPUs, *University of Tennessee Computer Science Technical Report*, July 29, 2010.

[8] NVIDIA Corporation. CUBLAS Library. NVIDIA Corporation, 2009. `http://www.nvidia.com/`

[9] NVIDIA Corporation. CUDA Programming Guide. NVIDIA Corporation, 2009. `http://www.nvidia.com/`

[10] P. Yalamov, D. J. Evans: The WZ matrix factorization method, *Parallel Computing* **21** (1995), pp. 1111–1120.

[11] `http://www.netlib.org/blas/`