High Performance Portable Solver for Tridiagonal Toeplitz Systems of Linear Equations

Beata Dmitruk and Przemysław Stpiczyński
Institute of Computer Science, Maria Curie-Skłodowska University, ul. Akademicka 9, 20-033 Lublin, Poland
beata.dmitruk@umcs.pl, przem@hektor.umcs.lublin.pl

Abstract. We show that recently developed divide and conquer parallel algorithm for solving tridiagonal Toeplitz systems of linear equations can be easily and efficiently implemented for a variety of modern multicore and GPU architectures, as well as hybrid systems. Our new portable implementation that uses OpenACC can be executed on both CPU-based and GPU-accelerated systems. More sophisticated variants of the implementation are suitable for systems with multiple GPUs and it can use CPU and GPU cores. We consider the use of both column-wise and row-wise storage formats for two dimensional double precision arrays and show how to efficiently convert between these two formats using cache memory. Numerical experiments performed on Intel CPUs and Nvidia GPUs show that our new implementation achieves relatively good performance.

Keywords: Tridiagonal Toeplitz systems · Parallel algorithms · Vectorization · Portability · OpenACC · Hybrid systems

1 Introduction

Tridiagonal Toeplitz systems of linear equations appear in many theoretical and practical applications. For example, numerical algorithms for solving boundary value problems for ordinary and partial differential equations reduce to such systems [13,15]. They also play an important role in piecewise cubic interpolation and spline algorithms [4,14]. There are several methods for solving such systems (the review of the literature can be found in [5]). Rojo [10] proposed a method for solving symmetric tridiagonal Toeplitz systems using LU decomposition of a system with almost Toeplitz structure together with Sherman-Morrison’s formula and this approach was modified to obtain new solvers for a possible parallel execution [7,9].

In [5] we proposed a new divide and conquer parallel algorithm for solving such systems using the splitting $T = LR + P$, where $L, R$ are bidiagonal and $P$ has only one non-zero entry. We showed how to reduce the number of necessary
synchronizations and use SIMD extensions of modern processors. Our OpenMP (version 3.1) implementation of this method achieved very good speedup on Intel Xeon CPUs (up to 5.06) and Intel Xeon Phi (up to 29.45). While this approach can be further improved using more sophisticated vectorization techniques such as the use of intrinsics [1,8,11], it will result in a loss of portability between different architectures. OpenACC is a standard for accelerated computing [2,6]. It offers compiler directives for offloading C/C++ and Fortran programs from host to attached accelerator devices. Such simple directives allow marking regions of source code for automatic acceleration in a portable vendor-independent manner. However, sometimes it is desired to apply some high-level transformations of source codes to achieve better performance [2,6,12]. Marked sources can be compiled for a variety of accelerators and parallel computers based on multicore CPUs. It is also possible to use OpenACC and OpenMP together to utilize CPU and GPU cores at the same time.

In this paper, we present a new portable OpenACC-based implementation of the algorithm that can be used on both CPUs and GPU accelerators without any changes in the source code. We also show how to use both OpenACC and OpenMP to utilize multiple GPUs, as well as implement the algorithm for hybrid systems. We study its performance on Intel Xeon CPUs and three Nvidia GPUs architectures: Kepler, Turing, and Volta. We consider both column-wise and row-wise storage formats for two dimensional arrays and show how to efficiently convert arrays between these two formats using cache memory to ensure coalesced access to device’s global memory. We also discuss which format is more suitable for CPU-based and GPU-accelerated architectures.

2 Parallel Algorithm

Let us consider a tridiagonal Toeplitz system of linear equations $Tx = f$ of the following form

$$
\begin{bmatrix}
  t_2 & t_3 \\
  t_1 & t_2 & t_3 \\
  \ddots & \ddots & \ddots \\
  t_1 & t_2 & t_3 \\
  t_1 & t_2 \\
\end{bmatrix}
\begin{bmatrix}
  x_0 \\
  x_1 \\
  \vdots \\
  \vdots \\
  x_{n-1} \\
\end{bmatrix}
= 
\begin{bmatrix}
  f_0 \\
  f_1 \\
  \vdots \\
  \vdots \\
  f_{n-1} \\
\end{bmatrix},
$$

(1)

For the sake of simplicity we assume that $n = 2^k$, $k \in \mathbb{N}$. The matrix $T$ can be decomposed as

$$
T =
\begin{bmatrix}
  1 & \alpha & 1 & \cdots & \cdots \\
  \alpha & 1 & \ddots & \cdots & \cdots \\
  \cdots & \cdots & \ddots & \cdots & \cdots \\
  \alpha & 1 & \cdots & 1 & \cdots \\
  \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}
\begin{bmatrix}
  \beta & t_3 \\
  \beta & t_3 & \ddots \\
  \ddots & \ddots & \ddots & \ddots \\
  \beta & t_3 \\
  \beta \\
\end{bmatrix}
+ 
\begin{bmatrix}
  t_3 & \alpha & 0 & \cdots & 0 \\
  0 & 0 & \cdots & \cdots & \cdots \\
  \vdots & \vdots & \ddots & \ddots & \cdots \\
  0 & \cdots & \cdots & \cdots & 0 \\
  0 & \ldots & \ldots & \ldots & 0 \\
\end{bmatrix},
$$

(2)
where $\alpha = (t_2 + \text{sign}(t_2)\sqrt{(t_2)^2 - 4t_1t_3})/(2t_3)$ and $\beta = t_2 - t_3\alpha$. Using (2) we can rewrite the Eq. (1) as follows

$$
\begin{bmatrix}
 x_0 \\
 x_1 \\
 \vdots \\
 x_{n-1}
\end{bmatrix} = (LR)^{-1}
\begin{bmatrix}
 f_0 \\
 f_1 \\
 \vdots \\
 f_{n-1}
\end{bmatrix} - t_3\alpha x_0 (LR)^{-1}
\begin{bmatrix}
 1 \\
 0 \\
 \vdots \\
 0
\end{bmatrix}
$$

or simply $\mathbf{x} = \mathbf{v} - t_3\alpha x_0 \mathbf{u}$. Then the solution to (1) can be found using:

$$
\begin{cases}
 x_0 = \frac{v_0}{1 + t_3^2\alpha u_0} \\
 x_i = v_i - t_3\alpha x_0 u_i, \quad i = 1, \ldots, n - 1.
\end{cases}
$$

In order to solve (1) we have to find two vectors $\mathbf{v}$ and $\mathbf{u}$, solving two systems of linear equations with the same coefficient matrix, namely $LR$. The solution to a system of linear equations $LR\mathbf{y} = \mathbf{d}$ can be found in two stages. First, we solve $L\mathbf{z} = \mathbf{d}$, and then $R\mathbf{y} = \mathbf{z}$. This can be easily done using the following simple sequential algorithm based on the two recurrence relations:

$$
\begin{cases}
 z_0 = d_0 \\
 z_i = d_i - \alpha z_{i-1}, \quad i = 1, \ldots, n - 1,
\end{cases}
$$

and

$$
\begin{cases}
 y_{n-1} = z_{n-1}/\beta \\
 y_i = (z_i - t_3y_{i+1})/\beta, \quad i = n - 2, \ldots, 0.
\end{cases}
$$

This simple sequential algorithm can be efficient only for small values of $n$ because it does not utilize the underlying hardware of modern processors, namely multiple cores and vector units. In order to obtain an efficient parallel algorithm for solving (5) let us consider the following divide and conquer method. First, we choose two integers $r, s > 1$, $rs = n$, and rewrite $L$ in the following block form:

$$
L = \begin{bmatrix}
 L_s & B & L_s \\
 B & L_s & \ddots & \vdots \\
 \vdots & \ddots & \ddots & \ddots \ \\
 B & L_s & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\end{bmatrix}, \quad L_s = \begin{bmatrix}
 1 \\
 \alpha & 1 \\
 \vdots \\
 \alpha & 1
\end{bmatrix}, \quad B = \begin{bmatrix}
 0 & \cdots & 0 & \alpha \\
 0 & \ddots & \vdots & \vdots \\
 \vdots & \ddots & 0 & \ddots \\
 0 & \cdots & \cdots & 0
\end{bmatrix}.
$$

Let us define:

$$
\mathbf{e}_k = (0, \ldots, 0, 1, 0, \ldots, 0)^T \in \mathbb{R}^s, \quad k = 0, \ldots, s - 1,
$$

and split $\mathbf{d}, \mathbf{z}$ into vectors $\mathbf{d}_i = (d_{is}, \ldots, d_{(i+1)s-1})^T$, $\mathbf{z}_i = (z_{is}, \ldots, z_{(i+1)s-1})^T \in \mathbb{R}^s$, for $i = 0, \ldots, r - 1$. Then $L\mathbf{z} = \mathbf{d}$ can be rewritten as follows:

$$
\begin{cases}
 z_0 = L_s^{-1}d_0 \\
 z_i = L_s^{-1}d_i - \alpha z_{i-1}L_s^{-1}\mathbf{e}_0, \quad i = 1, \ldots, r - 1.
\end{cases}
$$
Similarly, in case of the upper bidiagonal system \( Ry = z \), assuming the same as previously, we get the following block form of \( R \):

\[
R = \begin{bmatrix} R_s & C \\ \vdots & \ddots & \ddots \\ R_s & \cdots & C \\ R_s \\ \end{bmatrix}, \quad R_s = \begin{bmatrix} \beta t_3 \\ \vdots \\ \ddots \\ t_3 \\ \end{bmatrix}, \quad C = \begin{bmatrix} 0 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & \ddots \\ t_3 & 0 & \ldots & 0 \\ \end{bmatrix}.
\]

(9)

Analogously to (8), we get the following formula to find \( y \):

\[
\begin{cases}
    y_{r-1} = R_s^{-1}z_{r-1} \\
    y_i = R_s^{-1}z_i - t_3 y_{(i+1)s} R_s^{-1} e_{s-1}, \quad i = r - 2, \ldots, 0.
\end{cases}
\]

(10)

The algorithm for solving \( LRy = d \) based on (8) and (10) comprises two main stages (A and B), each consisting of three steps, has a lot of potential parallelisms. All vectors \( L^{-1}_s d_i, i = 0, \ldots, r - 1 \), can be found in parallel (Step A1). Indeed, when we form the matrix \( D = [d_0, \ldots, d_{r-1}] \), all columns of \( X = L^{-1}_s D \) can be found using the following vector-recursive formula:

\[
X_{i,*} \leftarrow D_{i,*} - \alpha \ast X_{i-1,*}, \quad i = 1, \ldots, s - 1,
\]

(11)

where \( X_{i,*} \) and \( D_{i,*} \) denote \( i \)-th rows of \( X \) and \( D \), respectively. Then we can find the last entry of each vector \( z_i, i = 1, \ldots, r - 1 \), sequentially (Step A2) applying (8). Finally, again in parallel, we calculate \( s - 1 \) remaining entries of \( z_1, \ldots, z_{r-1} \) (Step A3). In the case of (10) we proceed similarly. If \( Z = [z_0, \ldots, z_{r-1}] \), then \( Z \leftarrow R_s^{-1} Z \) can be found using:

\[
Z_{i,*} \leftarrow (Z_{i,*} - t_3 \ast Z_{i+1,*})/\beta, \quad i = s - 2, \ldots, 0.
\]

(12)

Then during the sequential part (Step B2) we find first entries of each \( y_i, i = r - 2, \ldots, 0 \), and finally (Step B3) we use (10) to calculate in parallel all remaining entries of \( Y = [y_0, \ldots, y_{r-1}] \). The algorithm should be applied twice to find vectors \( v \) and \( u \), respectively. Then we use (4) to find the solution to (2). Note that (4) can be easily vectorized and parallelized.

3 OpenACC-Based and Hybrid OpenMP+OpenACC Implementations

The algorithm for solving (2) presented in Sect. 2 can be easily implemented using OpenACC. Parallel steps A1, A3, B1, B3 can be vectorized and parallelized using parallel loop constructs. It means that the execution of the independent loops will be distributed among gangs that work in SIMD mode utilizing available hardware. The steps A2 and B2 are sequential and should be executed by a single gang. Figure 1 (left) shows our OpenACC implementation of the steps A1, A2, A3 using the column-wise storage format for the matrix \( X \) stored as a double precision array. This format is the most natural because there is no need to do
any data transfers after choosing specific values of r, s. Unfortunately, this format does not allow to use of coalesced memory access to global memory of devices [3] during the steps A1 and B1. Thus, one can expect that the performance of the implementation using such storage format can be much worse compared to the implementation using the row-wise storage format (Fig. 1, right). In this case, all references to global memory are coalesced. Figure 2 shows simple parallel loops that can be used to convert between column-wise storage and row-wise storage. Unfortunately, such conversion between formats can significantly reduce the overall performance because of non-coalesced memory access.

### Fig. 1. OpenACC implementations of the parallel algorithm based on (8) using column-wise storage (left) and row-wise storage (right)

```c
// column-wise storage
// Step A1
#pragma acc parallel present(x)
{
    #pragma acc loop independent
    for(int j=0;j<r;j++){
        for(int i=0;i<s;i++)
            x[j*s+i]=x[j*s+i-1]+alpha;
    }
}

// Step A2
#pragma acc parallel num_gangs(1)
    present(x)
{
    for(int j=0;j<r;j++)
        x[(j+1)*s-1]=x[j*s-1]+e0_last*alpha;
}

// Step A3
#pragma acc parallel deviceptr(e0)
    present(x)
{
    #pragma acc loop independent
    for(int i=0;i<s-1;i++)
        last1=x[j*s-1]+alpha;
    #pragma acc loop independent
    for(int j=0;j<r-1;j++)
        x[j*s+i]=last1+e0[i];
}
```

```c
// row-wise storage
// Step A1
#pragma acc parallel present(x)
{
    for(int i=1;i<s;i++){
        #pragma acc loop independent
        for(int j=0;j<r;j++)
            x[i*r+j]=x[(i-1)*r+j]+alpha;
    }
}

// Step A2
#pragma acc parallel num_gangs(1)
    present(x)
{
    for(int j=1;j<r;j++)
        x[n-r+j]=x[n-r+j-1]+e0_last*alpha;
}

// Step A3
#pragma acc parallel deviceptr(e0)
    present(x)
{
    #pragma acc loop independent
    for(int i=0;i<s-1;i++)
        last1=e0[i]+alpha;
    #pragma acc loop independent
    for(int j=1;j<r;j++)
        x[i*r+j]=x[(s-1)*r+j-1]+last1;
}
```

### Fig. 2. Simple conversion between considered column-wise and row-wise array storage formats using OpenACC parallel construct

```c
// from column-wise to row-wise
#pragma acc parallel loop independent present(b,x)
for(int i=0;i<n;i++)
    x[i]=b[(i/r)+(i%r)*r];
...

// from row-wise to column-wise
#pragma acc parallel loop independent present(b,x)
for(int i=0;i<n;i++)
    b[i]=x[(i/s)+(i%s)*r];
```
In order to improve the performance of the implementation let us consider a more sophisticated method for conversion between column-wise and row-wise storage formats that uses cache memory to ensure coalesced memory access (Fig. 5). Each gang of $BSIZE$ threads is responsible for reading a block of $VL \times BSIZE$ elements from the array stored column-wise using coalesced memory access and writing it to the cache memory. Then such a block is moved to a new array stored row-wise. Figures 3 and 4 explain how to do it. A gang of threads operates on a sequence of $VL$ blocks of $VL \times NC$ elements, where $VL$ is the size of a warp [3]. As soon as all blocks are in the cache memory, all threads within a gang write rows of $BSIZE$ elements into global memory. Devices coalesce such global memory access issued a warp into as few transactions as possible to minimize DRAM bandwidth [3]. Finally, it should be noted that in both cases (i.e. column-wise and row-wise storage formats) the steps B1, B2, B3 have been implemented similarly to excerpts of the source code shown in Fig. 1. Computations according to (4) have been implemented using acc parallel loop construct.

Figure 6 shows how to use OpenMP and OpenACC to utilize two GPUs. Two halves of the arrays are stored row-wise in global memories of GPUs. We create two OpenMP threads, each responsible for controlling one GPU. GPUs share data via host memory. The array shared_data is allocated on the host and each GPU has its copy of this array. The update self construct is used to update data on the host, while update device updates device memory. It is necessary to synchronize threads using the omp barrier construct. This approach can also be applied to utilize GPU and CPU cores at the same time. Figure 7 shows the idea of hybrid implementation using OpenMP and OpenACC. CPU is responsible for finding $r_1$ first columns, while GPU is used to find further $r_2$ columns, where $r_1 + r_2 = r$. Note that in this case, OpenMP nested parallelism should be enabled [8].
Conversion of a $VL \times BSIZE$ block stored column-wise to row-wise storage performed by a single gang of threads ($VL = 32$, $BSIZE = 256$)

```c
#pragma acc parallel present(b) deviceptr(x) vector_length(BSIZE)
{
  float xc[VL][BSIZE];
  #pragma acc cache(xc)

  for(int k=0;k<s;k+=VL){
    // each gang reads VL x BSIZE block from column-wise format to cache
    #pragma acc loop gang
    for(int j=0;j<r;j+=BSIZE){
      #pragma acc loop seq
      for(int l=0;l<BSIZE;l++NW)
        #pragma acc loop vector
        for(int i=0;i<BSIZE;i++)
          xc[i*VL][l+i/VL] = x[(j+l+i/VL)*s+k*i*VL];
    }
    // each gang writes VL x BSIZE block from cache to row-wise format
    #pragma acc loop gang vector
    for(int j=0;j<r;j+=)
      #pragma acc loop seq
      b[(k+i)*r+j] = xc[i][j*BSIZE];
  }
}
```

OpenACC implementation of the conversion from column-wise to row-wise using device cache memory
4 Results of Experiments

All experiments have been carried out on four different target architectures allowing OpenACC and OpenMP programming models. **Multicore** is a server with two Intel Xeon E5-2670 v3 (totally 24 cores with hyper-threading, 2.3 GHz), 128 GB RAM, running under Linux with CUDA 10.0 and Portland Group PGI compilers and tools version 19.4 with OpenMP and OpenACC support. **Kepler** is just like **multicore**, with NVIDIA Tesla K40m GPU (2880 cores, 12 GB RAM). Similarly, **Volta** is just like **multicore**, but with NVIDIA Tesla V100 GPU (5120 cores, 32 GB RAM). Finally, **Turing** is a server with Intel Core i7 (totally 4 cores with hyper-threading, 3.0 GHz), 24 GB RAM, NVIDIA GEFORCE RTX 2080 SUPER GPU (3072 cores, 8 GB RAM) running under Linux with CUDA 10.0 and PGI compilers.

We have tested three versions of our OpenACC-based implementation. The first one uses *column-wise* storage. The next two implementations rely on *row-wise* storage using the simple conversion between formats (Fig. 2) and the more sophisticated method (Fig. 5) that utilizes cache memory, respectively.
Table 1. Multicore: execution time for the best values of \( r \) and speedup of column-wise method over the sequential Thomas algorithm

| \( n \) | Thomas alg. r | Seq. alg. r | OpenMP r | Column-wise r | Row-wise r | Row-wise with cache r | Speedup |
|-------|--------------|------------|----------|--------------|-----------|----------------------|--------|
| 2     | 0.0011       | 0.0016     | 2\(^6\) 0.0005 2\(^5\) | 0.0007     | 2\(^9\) 0.0084 2\(^8\) | 0.0026   | 1.55    |
| 2     | 0.0020       | 0.0029     | 2\(^7\) 0.0008 2\(^6\) | 0.0009     | 2\(^10\) 0.0085 2\(^8\) | 0.0043   | 2.18    |
| 2     | 0.0039       | 0.0054     | 2\(^8\) 0.0014 2\(^6\) | 0.0013     | 2\(^10\) 0.0089 2\(^8\) | 0.0057   | 3.04    |
| 2     | 0.0077       | 0.0104     | 2\(^4\) 0.0025 2\(^7\) | 0.0019     | 2\(^11\) 0.0095 2\(^8\) | 0.0076   | 4.05    |
| 2     | 0.0148       | 0.0186     | 2\(^6\) 0.0032 2\(^8\) | 0.0026     | 2\(^11\) 0.0102 2\(^8\) | 0.0077   | 5.68    |
| 2     | 0.0284       | 0.0362     | 2\(^6\) 0.0048 2\(^8\) | 0.0040     | 2\(^11\) 0.0133 2\(^8\) | 0.0108   | 7.03    |
| 2     | 0.0564       | 0.0741     | 2\(^6\) 0.0092 2\(^8\) | 0.0076     | 2\(^11\) 0.0177 2\(^8\) | 0.0160   | 7.39    |
| 2     | 0.1134       | 0.1528     | 2\(^8\) 0.0199 2\(^10\) | 0.0179     | 2\(^11\) 0.0411 2\(^8\) | 0.0308   | 6.34    |
| 2     | 0.2249       | 0.3030     | 2\(^8\) 0.0431 2\(^10\) | 0.0405     | 2\(^12\) 0.0968 2\(^8\) | 0.0671   | 5.55    |
| 2     | 0.4478       | 0.6055     | 2\(^8\) 0.0879 2\(^10\) | 0.0865     | 2\(^14\) 0.1859 2\(^8\) | 0.1368   | 5.18    |
| 2     | 0.8939       | 1.2097     | 2\(^6\) 0.1512 2\(^11\) | 0.1728     | 2\(^14\) 0.3629 2\(^8\) | 0.2754   | 5.17    |
| 2     | 1.7859       | 2.4179     | 2\(^6\) 0.2809 2\(^13\) | 0.3480     | 2\(^14\) 0.7270 2\(^8\) | 0.5540   | 5.13    |
| 2     | 3.5695       | 4.8255     | 2\(^6\) 0.5385 2\(^14\) | 0.6983     | 2\(^13\) 1.4673 2\(^8\) | 1.0915   | 5.11    |
| 2     | 7.1368       | 9.6747     | 2\(^8\) 1.0314 2\(^14\) | 1.4021     | 2\(^14\) 2.9030 2\(^8\) | 2.1780   | 5.09    |
| 2     | 14.2987      | 19.3526    | 2\(^8\) 2.0027 2\(^15\) | 2.8156     | 2\(^15\) 5.8509 2\(^8\) | 4.3571   | 5.08    |
| 2     | 28.5449      | 38.7018    | 2\(^9\) 3.9222 2\(^15\) | 5.6348     | 2\(^15\) 11.6869 2\(^8\) | 8.8644   | 5.07    |

On multicore (Table 1) we have also tested the OpenMP version of column-wise implementation compiled with the PGI compiler, the sequential Thomas algorithm, the algorithm based on sequential formulas (5), (6) and the simple automatic parallelization of (4).

Tests have been performed for various problem sizes and values of the parameter \( r \). Because of global memory limitations, in case of Kepler and Turing, the biggest problem size is \( n = 2^{28} \). We have observed that the best performance is achieved when \( r \) is \( O(\sqrt{n}) \), and then the performance differs slightly. Tables 1, 2, and 3 show the results obtained for the best value of \( r \). Table 2 also contains the results obtained for our OpenMP+OpenACC implementations (i.e. for two GPUs and hybrid one utilizing GPU and CPU cores). Moreover, it shows the order of the relative error
Table 2. Kepler: relative error and the execution time for the best values of $r$. * - results obtained using the hybrid implementation

| $n$ | Error | $2 \times$ Kepler |
|-----|-------|------------------|
|     |       | Kepler           |                  |
|     |       | Column-wise      | Row-wise         | Row-wise with cache | Row-wise with cache |
|     |       | $r$ | Time | $r$ | Time | $r$ | Time | $r$ | Time |
| $2^{15}$ | $10^{-14}$ | $2^9$ | 0.0012 | $2^9$ | 0.0012 | $2^8$ | 0.0011 | $2^8$ | 0.0013 |
| $2^{16}$ | $10^{-14}$ | $2^9$ | 0.0018 | $2^{10}$ | 0.0016 | $2^9$ | 0.0016 | $2^9$ | 0.0016 |
| $2^{17}$ | $10^{-14}$ | $2^{10}$ | 0.0023 | $2^{10}$ | 0.0018 | $2^{10}$ | 0.0022 | $2^{10}$ | 0.0025 |
| $2^{18}$ | $10^{-15}$ | $2^{10}$ | 0.0032 | $2^{11}$ | 0.0025 | $2^{10}$ | 0.0030 | $2^{10}$ | 0.0031 |
| $2^{19}$ | $10^{-15}$ | $2^{10}$ | 0.0045 | $2^{11}$ | 0.0034 | $2^{10}$ | 0.0040 | $2^{10}$ | 0.0044 |
| $2^{20}$ | $10^{-16}$ | $2^{11}$ | 0.0062 | $2^{11}$ | 0.0047 | $2^{11}$ | 0.0054 | $2^{11}$ | 0.0056 |
| $2^{21}$ | $10^{-16}$ | $2^{11}$ | 0.0095 | $2^{12}$ | 0.0069 | $2^{11}$ | 0.0079 | $2^{12}$ | 0.0082 |
| $2^{22}$ | $10^{-16}$ | $2^{12}$ | 0.0152 | $2^{12}$ | 0.0108 | $2^{12}$ | 0.0115 | $2^{12}$ | 0.0112 |
| $2^{23}$ | $10^{-16}$ | $2^{12}$ | 0.0249 | $2^{13}$ | 0.0180 | $2^{12}$ | 0.0181 | $2^{12}$ | 0.0173 |
| $2^{24}$ | $10^{-15}$ | $2^{12}$ | 0.0481 | $2^{13}$ | 0.0334 | $2^{13}$ | 0.0296 | $2^{13}$ | 0.0241 |
| $2^{25}$ | $10^{-14}$ | $2^{12}$ | 0.0926 | $2^{14}$ | 0.0715 | $2^{13}$ | 0.0511 | $2^{13}$ | 0.0384 |
| $2^{26}$ | $10^{-14}$ | $2^{12}$ | 0.1938 | $2^{14}$ | 0.1636 | $2^{13}$ | 0.0950 | $2^{14}$ | 0.0607 |
| $2^{27}$ | $10^{-14}$ | $2^{15}$ | 0.4599 | $2^{14}$ | 0.3276 | $2^{14}$ | 0.1802 | $2^{14}$ | 0.1029 |
| $2^{28}$ | $10^{-13}$ | $2^{16}$ | 0.9259 | $2^{14}$ | 0.6757 | $2^{14}$ | 0.3614 | $2^{15}$ | 0.1856 |
| $2^{29}$ | $10^{-12}$ | – | – | – | – | $2^{16}$ | 1.2689$^*$ | $2^{15}$ | 0.3491 |
| $2^{30}$ | $10^{-11}$ | – | – | – | – | $2^{16}$ | 2.7807$^*$ | – | – |

achieved when solving the system of linear equations that arises for second order ordinary differential equations $y''(x) - py'(x) - qy(x) = g(x)$, where $x \in [a, b]$ and $y(a) = A, y(b) = B$.

We can observe that column-wise storage is the best for multicore and in this case, there is no need to perform conversion to row-wise storage. Thus, such conversion should be performed only if any GPU device is present. It can be easily checked using the `acc_get_num_devices()` function from OpenACC Runtime Library. It means that the source code will work properly on GPU-accelerated and non-accelerated systems without any changes.

Our OpenACC implementation compiled for non-accelerated multicore systems achieves good speedup over the sequential Thomas algorithm (up to 7.39) and its performance is about 65% of the performance achieved by our OpenMP-based implementation [5]. Thus, if we only consider CPU as the target architecture, the use of OpenMP is a better choice. In most cases for all tested GPUs, the implementation using row-wise storage achieves much better performance than the version using column-wise storage. We can also observe that the use of cache memory allows to speedup conversion between considered storage formats, especially on Kepler. In this case, the version using row-wise storage is more than 50% faster for bigger problem sizes. In the case of Turing and Volta, the use of
| n   | Turing                | Volta                |
|-----|-----------------------|----------------------|
|     | Column-wise          | Row-wise             | Row-wise with cache | Column-wise          | Row-wise             | Row-wise with cache |
|     | r   | Time      | r   | Time      | r   | Time      | r   | Time      | r   | Time      | r   | Time      |
| 2   | 10 | 0.0007   | 2   | 0.0007   | 2   | 0.0007   | 2   | 0.0006   | 2   | 0.0006   |
| 2   | 11 | 0.0009   | 2   | 0.0009   | 2   | 0.0009   | 2   | 0.0009   | 2   | 0.0009   |
| 2   | 12 | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   |
| 2   | 13 | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   |
| 2   | 14 | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   | 2   | 0.0010   |

Cache memory is profitable for bigger problem sizes. It should be noticed that the performance on all considered GPUs significantly outperforms the performance achieved on multicore. Thus, the use of our hybrid implementation is profitable for bigger problem sizes, i.e. when the GPU memory capacity is exceeded (see Table 2, the results marked with “∗”). Our implementation for two GPUs scales very well (see Table 2, 2 × Kepler) without significant performance overheads (Fig. 8). It should be noticed that the timing results do not take into account the time needed to copy data between host and GPUs. However, solving considered systems is, in most cases, a part of a larger problem. The implementation achieves the performance of 6.6 GFLOPS on multicore and 68.4 GFLOPS on Volta what is far from the peak performances of those architectures, but is normal for problems where the ratio of the number of memory references to the number of arithmetic operations is $O(1)$. Finally, let us observe that the relative error of the solution obtained by the parallel algorithm is acceptable.

## 5 Conclusions and Future Work

We have presented the new portable OpenACC-based implementation of the solver for tridiagonal Toeplitz systems of linear equations. Numerical experiments show that column-wise storage is the best for CPU-based architectures and it achieves good speedup (up to 7.39) over the sequential Thomas algorithm. In most cases for all tested GPUs, the implementation using row-wise storage...
achieves much better performance than the version using column-wise storage and the use of cache memory allows to improve its performance. Moreover, all considered GPUs outperform CPU-based systems. We have also shown how to use OpenMP and OpenACC together in order to obtain the implementation suitable for systems with multiple GPUs or hybrid systems. In the future, we plan to study the numerical properties of the method, especially its stability.

References

1. Amiri, H., Shahbahrami, A.: SIMD programming using intel vector extensions. J. Parallel Distrib. Comput. 135, 83–100 (2020). https://doi.org/10.1016/j.jpdc.2019.09.012. http://www.sciencedirect.com/science/article/pii/S074373151830813X
2. Chandrasekaran, S., Juckeland, G. (eds.): OpenACC for Programmers: Concepts and Strategies. Addison-Wesley, Boston (2018)
3. Cheng, J., Grossman, M., McKercher, T. (eds.): Professional CUDA C Programming. Wiley, Indianapolis (2014)
4. Chung, K.L., Yan, W.M.: Parallel B-spline surface fitting on mesh-connected computers. J. Parallel Distrib. Comput. 35, 205–210 (1996). https://doi.org/10.1006/jpdc.1996.0082
5. Dmitruk, B., Sttpiczyński, P.: Vectorized parallel solver for tridiagonal Toeplitz systems of linear equations. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K. (eds.) PPAM 2019. LNCS, vol. 12043, pp. 93–103. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43229-4_9
6. Farber, R. (ed.): Parallel Programming with OpenACC. Morgan Kaufmann, Cambridge (2017)
7. Garey, L., Shaw, R.: A parallel method for solving linear equations with tridiagonal Toeplitz coefficient matrices. Comput. Math. Appl. 42(1), 1–11 (2001). https://doi.org/10.1016/S0898-1221(01)00125-0
8. Jeffers, J., Reinders, J., Sodani, A.: Intel Xeon Phi Processor High-Performance Programming: Knights Landing edition. Morgan Kaufman, Cambridge (2016)
9. McNally, J.M., Garey, L., Shaw, R.: A communication-less parallel algorithm for tridiagonal Toeplitz systems. J. Comput. Appl. Math. 212, 260–271 (2008). https://doi.org/10.1016/j.cam.2006.12.001
10. Rojo, O.: A new method for solving symmetric circulant tridiagonal systems of linear equations. Comput. Math. Appl. 20, 61–67 (1990). https://doi.org/10.1016/0898-1221(90)90165-G
11. Sttpiczyński, P.: Language-based vectorization and parallelization using intrinsics, OpenMP, TBB and Cilk Plus. J. Supercomput. 74(4), 1461–1472 (2018). https://doi.org/10.1007/s11227-017-2231-3
12. Sttpiczyński, P.: Algorithmic and language-based optimization of Marsa-LFIB4 pseudorandom number generator using OpenMP, OpenACC and CUDA. J. Parallel Distrib. Comput. 137, 238–245 (2020). https://doi.org/10.1016/j.jpdc.2019.12.004
13. Sttpiczyński, P., Potiopa, J.: Solving a kind of boundary-value problem for ordinary differential equations using Fermi - the next generation CUDA computing architecture. J. Comput. Appl. Math. 236(3), 384–393 (2011). https://doi.org/10.1016/j.cam.2011.07.028
14. Terekhov, A.V.: A highly scalable parallel algorithm for solving Toeplitz tridiagonal systems of linear equations. J. Parallel Distrib. Comput. 87, 102–108 (2016). https://doi.org/10.1016/j.jpdc.2015.10.004

15. Vidal, A.M., Alonso, P.: Solving systems of symmetric Toeplitz tridiagonal equations: Rojo’s algorithm revisited. Appl. Math. Comput. 219, 1874–1889 (2012). https://doi.org/10.1016/j.amc.2012.08.030