Proceedings of the 18th Conference on Computer Science and Intelligence Systems pp. 1155–1160
DOI: 10.15439/2023F9640
ISSN 2300-5963 ACSIS, Vol. 35

Performance of Portable Sparse Matrix-Vector Product Implemented Using OpenACC

Kinga Stec, Przemysław Stpiczyński
0009-0008-6562-8954, 0000-0001-8661-414X
Maria Curie-Skłodowska University, Institute of Computer Science
Akademicka 9, 20-033 Lublin, Poland
Email: kingastec439@gmail.com, przemyslaw.stpiczynski@umcs.pl

***Abstract*—The aim of this paper is to study the performance of OpenACC implementations of sparse matrix-vector product for several storage formats: CSR, ELL, JAD, pJAD, and BSR, achieved on Intel CPU and NVIDIA GPU platforms to compare them with the performance of SpMV implementations using the BSR storage format provided by *Intel MKL* and *NVIDIA cuSPARSE* libraries. Numerical experiments show that vendor-provided BSR is the best format for CPUs but in the case of GPUs, the pJAD storage format allows to achieve better performance.**

I. Introduction

SPARSE matrix-vector product (SpMV) is a central part of many numerical algorithms and its performance can have a very big impact on the performance of scientific and engineering applications [1], [2]. There are a lot of various sparse matrix storage formats and sophisticated techniques for developing efficient implementations of SpMV that utilize the underlying hardware of modern multicore CPUs and GPUs [3], [4], [5], [6], [7], [8], [9]. Unfortunately, these methods are rather complicated and usually depend on particular computer architecture, thus developing efficient and portable sparse matrix source code is still a challenge. However, the results presented in [10] and [11] show that simple SPARSKIT SpMV routines using various storage formats (CSR, ELL, JAD) [1] can be easily and efficiently adapted to modern CPU-based or GPU-accelerated architectures. Loops in source codes can be easily parallelized using OpenMP [12] or OpenACC [13], [14] directives, while the rest of the work can be done by a compiler. Such parallelized SpMV routines achieve performance comparable with the performance of the SpMV routines available in libraries optimized by hardware vendors (i.e. Intel MKL, NVIDIA cuSPARSE). OpenACC, a standard for accelerated computing, provides compiler directives for offloading C/C++ 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. Moreover, OpenACC programs can be compiled using the `multicore` option, and then such programs can also be run on CPU-based architectures [15], [16], [17] without any changes in source codes.

Recently, the Block Compressed Row (BSR) format [18], [19], which is a generalization of the Compressed Sparse Row (CSR) format, has become very popular. Intel MKL and NVIDIA cuSPARSE provide optimized SpMV implementations for this format. Moreover, the other formats have been deprecated. Especially, BSR has replaced the HYB format in cuSPARSE. In this paper we compare the performance of portable OpenACC implementations of sparse matrix-vector product for CSR, ELL, JAD, pJAD, and BSR with the performance of SpMV implementations using the BSR storage format provided in *Intel MKL* and *NVIDIA cuSPARSE* libraries.

II. Sparse Matrix Representations

Let us assume that A is a sparse matrix with a significant number of zero entries, and $\mathbf{x}$, $\mathbf{y}$ are dense vectors. The SpMV operation is defined as follows:

$$\mathbf{y} \leftarrow A\mathbf{x}. \tag{1}$$

It is clear that if we do not multiply entries of $\mathbf{x}$ by zero entries of A, then (1) requires $2 \cdot n_{nz}$ floating point operations (one multiplication and one addition per nonzero entry of A). The structure of a sparse matrix can be characterized by n, n_{nz}, n_{nz}/n, and $\max_{nz}$, where n is the number of rows, n_{nz} is the total number of nonzero elements, n_{nz}/n the average number of nonzero elements per row, $\max_{nz}$ is the biggest number of nonzero elements per row. Table I shows values of these parameters for a set of test matrices, selected from *Matrix Market* [20] and *University of Florida Sparse Matrix Collection* [21]. It is clear that the performance of SpMV depends on the matrix storage format that utilizes the underlying hardware.

For description purposes of several possible sparse matrix storage formats, let us consider the following matrix as an example:

$$A = \begin{bmatrix} 7 & 0 & 1 & 0 \\ 0 & 4 & 2 & 3 \\ 1 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}, \tag{2}$$

where $n = 4$, $n_{nz} = 8$, $n_{nz}/n = 2$, and $\max_{nz} = 3$. Now let us consider a few basic (ELL, JAD, CSR [1], [22]), as well as, more sophisticated (pJAD [11], BSR [18], [19]) storage formats for sparse matrices.

TABLE I: Set of test matrices [11]

Matrix	n	n_{nz}	n_{nz}/n	$\max_{nz}$
cry10000	10000	49699	5.0	5
possion3Da	13514	352762	26.1	110
af23560	23560	484256	20.6	21
g7jac140	41490	565956	13.6	153
fidapm37	9152	765944	83.7	255
bcsstk36	23052	1143140	49.6	178
majorbasis	160000	1750416	10.9	11
bbmat	38744	1771722	45.7	126
cfd1	70656	1828364	25.9	33
ASIC_680ks	682712	2329176	3.4	210
FEM_3D_thermal2	147900	3489300	23.6	27
parabolic_fem	525825	3674625	7.0	7
ecology2	999999	4995991	5.0	5
pre2	659033	5959282	9.0	628
boneS01	127224	6715152	52.8	81
torso1	116158	8516500	73.3	3263
thermal2	1228045	8580313	7.0	11
atmosmodl	1489752	10319760	6.9	7
bmw3_2	227362	11288630	49.7	336
af_shell8	504855	17588875	34.8	40
cage14	1505785	27130349	18.0	41
nd24k	72000	28715634	398.8	520
inline_1	503712	36816342	73.1	843
ldoor	952203	46522475	48.9	77
cage15	5154859	99199551	19.2	47

A. ELL

The ELL storage format was introduced in *Ellpack-Itpack* package. It assumes that a sparse matrix is represented by two arrays (Figure 1). Nonzero elements are stored in the first one called `a`. The second one called `ja` contains the corresponding column indices [23]. Both arrays are $n \times ncol$, where $ncol = \max_{nz}$. While ELL is simple and provides easy access to matrix entires, when $n_{nz}/n \ll \max_{nz}$, the number of stored zero entries of the matrix increases significantly.

`a:`

7	1	*
4	2	3
1	8	*
9	*	*

`ja:`

0	2	*
1	2	3
0	1	*
1	*	*

Fig. 1: ELL format for (2)

B. JAD

The JAD (i.e. *Jagged Diagonal*) format storage is represented by three arrays (Figure 2). It is similar to ELL, but removes the assumption on the fixed-length rows [22]. Firstly, a sparse matrix needs to be sorted in non-increasing order of the number of nonzeros per row

$$PA = \begin{bmatrix} 0 & 4 & 2 & 3 \\ 7 & 0 & 1 & 0 \\ 1 & 8 & 0 & 0 \\ 0 & 9 & 0 & 0 \end{bmatrix}.$$

The arrays `a` and `ja` of dimension nz contain nonzero elements (i.e. jagged diagonals) and the corresponding column indices. The array `ia` contains the beginning position of each jagged diagonal. Additionally, we can add array `rlen` which contains the number of nonzero elements in each row. Entries of this array can be calculated (in parallel) using the following formula. Let $jdiag$ be the number of jagged diagonals. Then for each row, $i = 0, \ldots, n-1$, we have

$$\mathbf{rlen}[i] = |\{j : 0 \le j \le jdiag - 1 \wedge \mathbf{ia}[j+1] - \mathbf{ia}[j] > i\}|.$$

Note that this format is devoid of the inconvenience associated with the need to store zero elements in rows completed to the width of $\max_{nz}$.

`a:`

4	2	3
7	1	
1	8	
9		

`ia:`

0
4
7
8

`rlen:`

3
2
2
1

`ja:`

1	2	3	0	2	0	1	2

Fig. 2: JAD format for (2)

C. pJAD

The pJAD storage format is an optimized version of JAD (Figure 3). This format assumes aligning (padding) columns of the arrays `a` and `ja` [11]. We add zero elements, thus the number of elements of each column should be a multiple of a given $bsize$ and rows of each block should have the same length. Entries of the array `brlen` contain widths of blocks of $bsize$ rows. Note that pJAD assumes to store at most $jdiag \cdot (bsize - 1)$ additional zero entries, where $jdiag$ is the number of jagged diagonals stored in `a`. Padding of jagged diagonals is important especially for GPUs. It allows *coalesced memory access* and reduces *thread divergence* within a block of threads [24].

`a:`

4	2	3
7	1	0
1	8	
9	0	

`ia:`

0
4
8
10

`brlen:`

3
2

`ja:`

1	2	3	0	2	0	1	*	2	*

Fig. 3: pJAD format for (2)

D. CSR

A sparse matrix in CSR (i.e. *Compressed Sparse Rows*) is stored in three arrays (Figure 4). The first array called `data` contains nonzero elements, and the second one called `cols` contains corresponding column indices of nonzero values. Indices of the beginning of rows in `data` array are stored at the `ptr` array.

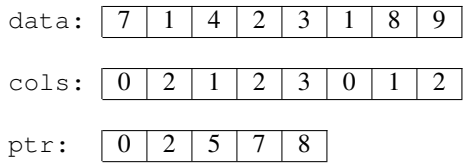

Fig. 4: CSR format for (2)

E. BSR

The BSR storage format can be treated as a generalization of CSR. A sparse matrix is represented by four arrays (Figure 5). Array `vals` contains column ordered values from blocks with nonzero values. Array `cols` stored columns indices of the first element per block. The `ptrB` and `ptrE` arrays contain the indices of the beginning and ending positions of the elements in the block row respectively.

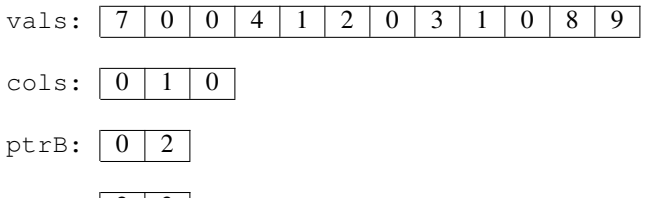

Fig. 5: BSR format for (2)

III. Algorithms

The SpMV operation for all storage formats presented in Section II can be implemented using OpenACC to be executed on both GPU-accelerated and CPU-based systems. OpenACC offers compiler directives for offloading selected computations from host to attached accelerator devices. It allows to indicate regions of source code for automatic parallelization in a portable manner. Algorithms 1, 2, 3, 4, and 5 show how to implement SpMV in C/C++ using OpenACC for all considered formats: ELL, JAD, pJAD, CSR, and BSR formats, respectively. OpenACC-specific parts of the implementation start with `#pragma acc` directives. The `parallel loop` directive defines a loop to be accelerated on GPU. Additional clauses, namely `gang` and `vector_length` tell that `gangs` (i.e. blocks of threads) should perform an iteration of loops. Threads within gangs work in vector or SIMD mode [13]. The `loop seq` construct placed before a loop within `parallel loop` says that such a loop should be executed sequentially by a single thread. The `present` clause says that indicated variables are previously allocated on GPU. It allows to avoid unnecessary data movements between host and device memory systems. OpenACC provides the `data` construct that can be used to specify such scope of data in accelerated regions. Data transfers can also be initialized using the `enter data` and `exit data` constructs [13]. Figure 6 shows output messages generated by the compiler using the `-acc=gpu` option.

When OpenACC programs are compiled using the `-acc=multicore` option, the compiler generates appropriate parallel regions to be executed in parallel on CPU cores (Figure 7). It should be noticed that if we omit OpenACC directives, we will get sequential implementations of SpMV.

Algorithm 1 SpMV using ELL in OpenACC

```
// auxiliary routine
double count_per_row(double *a, double *x, int *ja,
    int n,  int ncol, int i){
  double t = 0;
  #pragma acc loop seq
  for( int j = 0; j < ncol; ++j ) {
      t += a[j*n+i] * x[ja[j*n+i]];
  }
  return t;
}

// driver routine
void ELL_SpMV(int n, double *x, double *y, int ncol,
     double *a, int *ja){
  #pragma acc parallel loop gang vector_length(128)\
       present(y,x,a,ja)
  for(int i=0; i<n; i++) {
    y[i] = count_per_row(a, x, ja, n, ncol, i);
  }
}
```

Algorithm 2 SpMV using JAD in OpenACC

```
// auxiliary routine
double count_per_row(int rlen, int *ia, int i,
    double *a, double *x, int *ja){
  double t = 0;
  #pragma acc loop seq
  for(int j = 0; j<rlen; j++){
    int k = ia[j]+i;
    t+=a[k]*x[ja[k]];
  }
  return t;
}

// driver routine
void JAD_SpMV(int n, int *perm, double *a, int *rlen
    , int *ia, int *ja, double *x, double *y){
  #pragma acc parallel loop gang vector_length(128)\
       present(perm,a,rlen, ia, ja, x, y)
  for(int i = 0; i<n; i++){
    y[perm[i]] = count_per_row(rlen[i], ia, i, a, x,
         ja);
  }
}
```

IV. Performance of SpMV

All OpenACC implementations of SpMV have been tested on the computer equipped with two Xeon Gold 6342 @ 2.80GHz (48 cores) and NVIDIA A40 GPU (10752 cores, FP64 Peak perf. 584.6 GFLOPS), running under Linux Oper-

Algorithm 3 SpMV using pJAD in OpenACC

```
// auxiliary routine
double count_per_row(double *a, double *x, int *ia,
    int *ja, int brlen, int bsize, int i){
  double t = 0;
  #pragma acc loop seq
  for( int j = 0; j < brlen; ++j ) {
    int k = ia[j]+i;
    t += a[k]* x[ja[k]];
  }
  return t;
}

// driver routine
void pJAD_SpMV(int n_block, double *x, double *y,
    double *a, int *ja, int *ia, int *brlen, int *
    iperm, int bsize){
  #pragma acc parallel loop gang vector_length(128)\
       present(y,x,a,ja,ia,brlen,iperm)
  for(int i=0; i<n_block; i++) {
    #pragma acc loop
    for (int j=0; j<bsize; j++){
      y[iperm[i*bsize+j]] = count_per_row(a, x, ia,
          ja, brlen[i], bsize, i*bsize+j);
    }
  }
}
```

Algorithm 4 SpMV using CSR in OpenACC

```
// auxiliary routine
double count_per_row(int nz_in_row, int idx_start,
    int *cols, double *data, double *x){
  double t = 0;
  #pragma acc loop seq
  for(int j = 0; j<nz_in_row; j++){
     t+=x[cols[idx_start+j]]*data[idx_start+j];
  }
  return t;
}

// driver routine
void CSR_SpMV(int n, double *data, int *cols, int *
    ptr, double *x, double *y){
  #pragma acc parallel loop gang vector_length(128)\
       present(ptr,cols,x,y,data)
  for(int i = 0; i<n; i++){
    y[i] = count_per_row(ptr[i+1]-ptr[i],ptr[i],cols
        ,data,x);
  }
}
```

```
count_per_row:
   4, Generating implicit acc routine seq
      Generating acc routine seq
      Generating NVIDIA GPU code
ELL_SpMV:
  13, Generating present(y[:],x[:],ja[:],a[:])
      Generating NVIDIA GPU code
      15, #pragma acc loop gang, vector(128)
                    /* blockIdx.x threadIdx.x */
```

Fig. 6: Compiler output messages for Algorithm 1 compiled using `-acc=gpu`

Algorithm 5 SpMV using BSR in OpenACC

```
// auxiliary routine
void count_per_block(int block_size, int rows_begin,
    int rows_end,int *cols, double *vals ,double *x,
    double *y, int i){
  #pragma acc loop seq
  for(int j=rows_begin;j<rows_end;j++){
    int base=j*block_size*block_size;
    for(int jdx=cols[j]*block_size;jdx<(cols[j]+1)*
        block_size;jdx++){
          for(int idx=i*block_size; idx<(i+1)*
              block_size; idx++){
            y[idx]+=vals[base]*x[jdx];
            base++;
          }
    }
  }
}

// driver routine
void  BSR_SpMV(int rows,int block_size, int *ptrB,
    int *ptrE, int *cols, double *vals, double *x,
    double *y){
  #pragma acc parallel loop gang vector_length(128)\
       present(x,y,vals,ptrB,ptrE,cols)
  for(int i=0;i<rows;i++){
    count_per_block(block_size, ptrB[i], ptrE[i],
        cols, vals, x, y, i);
  }
}
```

```
ELL_SpMV:
     13, Generating Multicore code
         15, #pragma acc loop gang
```

Fig. 7: Compiler output messages of Algorithm 1 compiled using `-acc=multicore`

ating System with Intel OneAPI and NVIDIA HPC compiler suits. The results have been compared with SpMV implementations using the BSR storage format that are provided in *Intel MKL* and *NVIDIA cuSPARSE* libraries. Table II shows the performance (GFLOPS) obtained for all considered implementations for both CPUs and GPU and the set of sparse matrices from Table I calculated as follows:

$$perf = \frac{2 \cdot n_{nz}}{t \cdot 10^9} \text{ GFLOPS}, \tag{3}$$

where t is the execution time of SpMV (in seconds). It should be noticed that in the case of BSR, the table shows the best performance achieved for the optimal block size determined empirically. All experiments have been performed for FP64.

V. Results of Experiments

On CPU, the best performance for the majority of matrices is obtained for *Intel MKL* BSR implementation. For the smaller matrices, the best results are achieved by OpenACC implementation of SpMV using the CSR format. Other OpenACC implementations achieve worse performance than *Intel MKL* BSR. Especially, OpenACC BSR is much slower than its well-optimized counterpart. In most cases pJAD achieves better performance than JAD, however its performance is

TABLE II: SpMV performance results (GFLOPS)

Matrix	OpenACC CPU					MKL	OpenACC GPU					cuSPARSE
	CSR	ELL	JAD	pJAD	BSR	BSR	CSR	ELL	JAD	pJAD	BSR	BSR
cry10000	**2.56**	1.07	1.16	1.02	1.56	0.71	5.68	**6.14**	5.21	5.28	4.29	1.06
poisson3Da	**7.57**	1.43	6.85	6.37	3.62	3.21	**18.21**	12.47	14.95	16.60	3.28	3.82
af23560	**7.78**	7.59	5.13	4.76	1.12	4.68	26.74	**32.04**	29.02	29.76	22.45	9.45
g7jac140	**8.47**	0.97	2.65	3.11	0.75	3.99	**24.47**	6.62	18.32	20.85	6.02	6.22
fidapm37	**14.36**	3.31	7.27	6.91	8.63	7.04	**26.75**	20.76	18.46	21.85	5.82	10.44
bcsstk36	**12.18**	2.40	6.46	7.47	2.41	10.23	**33.30**	19.44	29.40	32.67	18.21	17.65
majorbasis	**12.51**	11.05	8.38	9.35	0.85	11.09	41.19	51.85	50.34	**51.89**	10.60	14.82
bbmat	**13.41**	4.14	5.59	6.09	2.49	12.58	40.97	25.93	47.80	**52.15**	22.71	22.49
cfd1	**14.22**	8.38	8.50	8.89	1.55	11.71	32.97	46.88	53.83	**55.38**	16.61	15.76
ASIC_680ks	4.25	0.21	2.07	2.20	0.52	**4.88**	**37.88**	1.42	23.69	25.91	9.40	12.53
FEM_3D_thermal2	14.27	9.76	6.81	9.17	1.49	**16.54**	37.78	56.75	60.18	**64.30**	14.63	23.79
parabolic_fem	**6.15**	5.73	5.45	5.68	0.82	5.76	44.07	50.83	48.88	**51.46**	4.13	15.39
ecology2	5.86	6.48	5.43	5.67	0.92	**7.61**	59.81	**63.25**	56.38	60.11	8.59	29.02
pre2	6.99	0.13	2.34	3.14	1.24	**7.35**	**45.28**	1.18	26.94	28.47	6.16	16.49
boneS01	14.03	7.98	5.84	8.91	3.97	**24.22**	38.42	50.39	72.20	**74.71**	22.06	41.06
torso1	12.07	0.27	0.71	0.52	4.23	**20.15**	**31.94**	1.82	21.03	23.66	6.31	30.01
thermal2	6.19	5.36	4.54	5.43	1.43	**7.17**	44.80	**46.54**	26.04	27.73	4.17	22.74
atmosmodl	7.73	8.11	6.88	6.93	1.69	**10.38**	61.82	**72.87**	58.44	62.32	7.00	32.63
bmw3_2	10.83	1.53	5.14	9.73	3.54	**27.16**	36.42	12.38	63.03	**68.40**	15.74	47.84
af_shell8	4.24	4.98	4.86	6.21	3.30	**21.26**	38.86	71.83	74.69	**79.50**	20.64	64.34
cage14	6.29	2.83	4.12	3.68	3.53	**11.17**	43.37	34.83	45.00	**50.33**	5.18	29.19
nd24k	10.64	7.00	6.31	5.38	11.38	**28.06**	23.83	68.88	88.65	**90.49**	30.91	78.49
inline_1	9.95	0.72	6.08	5.72	5.35	**20.26**	33.21	8.17	57.97	**64.76**	9.82	41.29
ldoor	9.42	4.23	5.25	6.22	7.14	**25.91**	39.40	51.67	51.54	58.89	16.55	**67.83**
cage15	10.50	3.68	5.60	6.40	4.57	**12.72**	**41.34**	31.92	34.23	37.62	4.70	27.75

still worse than the optimized BSR provided by Intel. The ELL format gives the worse performance for matrices with $n_{nz}/n \ll \max_{nz}$, when the number of stored zero entries increases significantly.

On GPU, we can observe that pJAD implementation achieves the best results for the largest number of matrices (eleven matrices). It outperforms JAD format for all matrices and *Nvidia cuSPARSE* BSR for almost all matrices. Moreover, for several matrices pJAD outperforms *Intel MKL* BSR significantly. The second best format is CSR. It gains the best results for seven matrices. For the others, the pJAD format is always better and the JAD format is almost always better. The ELL format achieves best results for five matrices (*cry10000, af23560, ecology2, thermal2, atmosmodl*), most of which have almost the same row length ($n_{nz}/n \approx \max_{nz}$). *Nvidia cuSPARSE* BSR gains the highest performance for only one matrix (i.e. *ldoor*), however for larger matrices it is much faster than OpenACC BSR. As with CPU, the difference between OpenACC and *Nvidia cuSPARSE* implementations of BSR is significant.

VI. Conclusions and Future Work

We have shown that sparse matrix-vector product using several formats can easily be implemented using OpenACC in order to utilize underlying hardware of modern CPUs and GPUs. Our implementations achieve reasonable performance on GPU and CPU, in some cases comparable with the performance of vendor optimized implementations using the BSR format, and sometimes even better.

It seems that the use of pJAD is very promising for GPUs. Its OpenACC portable implementation achieves much better performance than BSR optimized by the vendor. In the future we plan to provide non-portable optimized version of SpMV using pJAD.

References

[1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[2] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *J. Computational Applied Mathematics*, vol. 236, no. 15, pp. 3584–3590, 2012. doi: 10.1016/j.cam.2011.04.025

[3] X. Feng, H. Jin, R. Zheng, Z. Shao, and L. Zhu, "A segment-based sparse matrix-vector multiplication on CUDA," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 271–286, 2014. doi: 10.1002/cpe.2978

[4] J. C. Pichel, J. A. Lorenzo, F. F. Rivera, D. B. Heras, and T. F. Pena, "Using sampled information: is it enough for the sparse matrix-vector product locality optimization?" *Concurrency and Computation: Practice and Experience*, vol. 26, no. 1, pp. 98–117, 2014. doi: 10.1002/cpe.2949

[5] F. Vázquez, G. O. López, J. Fernández, and E. M. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," in *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, 2010. doi: 10.1109/CIT.2010.208 pp. 1146–1151.

[6] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009. doi: 10.1016/j.parco.2008.12.006

[7] K. K. Matam and K. Kothapalli, "Accelerating sparse matrix vector multiplication in iterative methods using GPU," in *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*, 2011. doi: 10.1109/ICPP.2011.82 pp. 612–621.

[8] C. Stylianou and M. Weiland, "Exploiting dynamic sparse matrices for performance portable linear algebra operations," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 2022. doi: 10.1109/P3HPC56579.2022.00010 pp. 47–57.

[9] B. Yilmaz, "A novel graph transformation strategy for optimizing SpTRSV on CPUs," *Concurrency and Computation Practice and Experience*, 2023. doi: 10.1002/cpe.7761

[10] B. Bylina, J. Bylina, P. Stpiczyński, and D. Szałkowski, "Performance analysis of multicore and multinodal implementation of SpMV operation," in *Proceedings of the Federated Conference on Computer Science and Information Systems, September 7-10, 2014, Warsaw, Poland*. IEEE, 2014. doi: 10.15439/2014F313 pp. 575–582.

[11] P. Stpiczyński, "Semiautomatic acceleration of sparse matrix-vector product using OpenACC," in *Parallel Processing and Applied Mathematics, 11th International Conference, PPAM 2015, Kraków, Poland, September 6-9, 2015, Revised Selected Papers, Part II*, ser. Lecture Notes in Computer Science, vol. 9574. Springer, 2016. doi: 10.1007/978-3-319-32152-3_14 pp. 143–152.

[12] R. van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP – The Next Step. Affinity, Accelerators, Tasking, and SIMD*. Cambridge MA: MIT Press, 2017.

[13] S. Chandrasekaran and G. Juckeland, Eds., *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley, 2018.

[14] R. Farber, Ed., *Parallel Programming with OpenACC*. Morgan Kaufmann, 2017.

[15] H. J. Eberl and R. Sudarsan, "OpenACC parallelisation for diffusion problems, applied to temperature distribution on a honeycomb around the bee brood: A worked example using BiCGSTAB," in *Parallel Processing and Applied Mathematics - 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part II*, 2013. doi: 10.1007/978-3-642-55195-6_29 pp. 311–321.

[16] P. Stpiczyński, "Algorithmic and language-based optimization of Marsa-LFIB4 pseudorandom number generator using OpenMP, OpenACC and CUDA," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 238–245, 2020. doi: 10.1016/j.jpdc.2019.12.004

[17] B. Dmitruk and P. Stpiczyński, "Solving tridiagonal Toeplitz systems of linear equations on GPU-accelerated computers," *Concurrency and Computation Practice and Experience*, vol. 34, p. 6449, 2022. doi: 10.1002/cpe.6449

[18] R. W. Vuduc and H. J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications, First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3726. Springer, 2005. doi: 10.1007/11557654_91 pp. 807–816.

[19] R. Shahnaz and A. Usman, "Blocked-based sparse matrix-vector multiplication on distributed memory parallel computers," *Int. Arab J. Inf. Technol.*, vol. 8, no. 2, pp. 130–136, 2011.

[20] R. F. Boisvert, R. Pozo, K. A. Remington, R. F. Barrett, and J. Dongarra, "Matrix Market: a web resource for test matrix collections," in *Quality of Numerical Software - Assessment and Enhancement, Proceedings of the IFIP TC2/WG2.5 Working Conference on the Quality of Numerical Software, Assessment and Enhancement, Oxford, UK, 8-12 July 1996*, ser. IFIP Conference Proceedings, R. F. Boisvert, Ed., vol. 76. Chapman & Hall, 1997, pp. 125–137.

[21] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011. doi: 10.1145/2049662.2049663

[22] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013. doi: 10.1007/s11227-012-0825-3

[23] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 user's guide," Center for Numerical Analysis, University of Texas, Tech. Rep. CNA-150, 1979.

[24] J. Cheng, M. Grossman, and T. McKercher, Eds., *Professional CUDA C Programming*. Wiley and Sons, 2014.