Journal of Computer and Communications, 2020, 8, 44-58
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

# PELLR: A Permutated ELLPACK-R Format for SpMV on GPUs

**Zhiqi Wang**[1], **Tongxiang Gu**[2*]

[1]School of Mathematics and Statistics, Northeast Normal University, Changchun, China
[2]Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, Beijing, China
Email: wangzq537@nenu.edu.cn, *txgu@iapcm.ac.cn

**How to cite this paper:** Wang, Z.Q. and Gu, T.X. (2020) PELLR: A Permutated ELLPACK-R Format for SpMV on GPUs. *Journal of Computer and Communications*, **8**, 44-58.
https://doi.org/10.4236/jcc.2020.84004

**Received:** March 6, 2020
**Accepted:** April 12, 2020
**Published:** April 15, 2020

## Abstract

The sparse matrix vector multiplication (SpMV) is inevitable in almost all kinds of scientific computation, such as iterative methods for solving linear systems and eigenvalue problems. With the emergence and development of Graphics Processing Units (GPUs), high efficient formats for SpMV should be constructed. The performance of SpMV is mainly determinted by the storage format for sparse matrix. Based on the idea of JAD format, this paper improved the ELLPACK-R format, reduced the waiting time between different threads in a warp, and the speed up achieved about 1.5 in our experimental results. Compared with other formats, such as CSR, ELL, BiELL and so on, our format performance of SpMV is optimal over 70 percent of the test matrix. We proposed a method based on parameters to analyze the performance impact on different formats. In addition, a formula was constructed to count the computation and the number of iterations.

## Keywords

SpMV, GPU, Storage Format, High Performance

## 1. Introduction

The Sparse matrix vector multiplication (SpMV) is a key operation in for a variety of computation science, such as in many iterative methods for solving linear systems ( $Ax = b$ ), image processing, simulation and so on. It is very important to improving the performance of the SpMV.

GPU including many Stream Processors, and many threads can simultaneously calculate multiple groups of data, with high computational power and very high memory bandwidth. It has high parallelism. GPU has many different types of memory, such as shared memory, texture memory, global memory, local

*Corresponding author.

memory and so on. Different memory access speed is also different, our computing will be greatly improved if we reasonably use them. The GPU architecture and CUDA programming model can see in [1] [2] [3]. Dense operations are regular and included by CUBLAS library [4]. However, the matrices obtained in engineering that needs to be actually calculated are mostly sparse and have no special format. The dimensions of the matrix are large (e.g. $>10^5$), with non-zero elements components is low (e.g. $\leq 5\%$). In order to improve the computational efficiency, it is important to make changes to find a suitable matrix storage format and calculation method.

There are many storage formats related to sparse matrix, such as CSR, ELL, HYB, BiELL and so on. In [5] we can see ELL performance for the structured matrices because it has continuous access to memory. The ELLPACK-R format presented in [6] is optimized to reduce the waiting time between different threads. The jagged diagonals (JAD) format was used to implement the SpMV Kernel in [7] to achieve a better performance. The BiELL format in [8] uses the bisection technique to improve the performance of ELL format. You can see more different formats in [9] [10] [11] [12].

This paper is an optimization of ELLPACK-R format, we call it PELLR. The remainder of the paper is organized as follows. Section 2 gives some existing sparse matrix storage formats. Related works and our new PELLR format are described in Section 3, and Section 4 presents some numerical results. The conclusions are stated in Section 5.

## 2. Basic Formats to Sparse Matrices

In this section, some basic sparse matrix storage formats are described. For a clearer understanding, let's use a simple model. A sparse matrix $A$ is represented by **Figure 1**. The white box represents zero elements, and the blue boxes represent non-zero elements.

### 2.1. COO Format

Coordinate (COO) storage format is the most direct and simple scheme for a

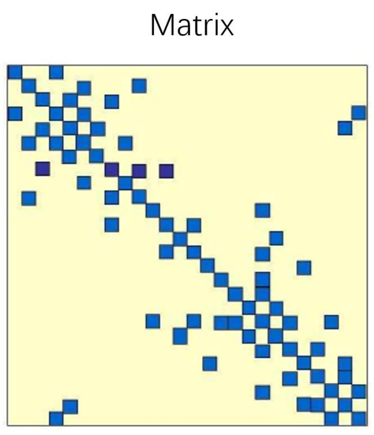

**Figure 1.** A sparse matrix: $A$ [6].

sparse matrix [13]. Let $nnz$ be the total number of non-zero entries of the matrix. Then COO can be represented by three one dimension arrays with the size of $nnz$.

- Real array $A[\,]$ contains the non-zero entries row by row in any order.
- Integer array $J[\,]$ is made of the corresponding column indices for each non-zero entry in $A[\,]$.
- Integer array $I[\,]$ is made of the corresponding row indices for each non-zero entry in $A[\,]$.

The calculation of SpMV based on COO format is not suitable for GPU structure when the matrix is stored in disorder. In this case, the multi-threads will access data and write vector in discontinuous way. On the other hand, this format will occupy more memory than that of CSR format, which will be introduced in next subsection.

## 2.2. CSR Format

The compressed sparse row (CSR) format is the most practical format to store sparse matrices [13]. It can also represent as three one dimension arrays. Let $N$ and $nnz$ be the number of rows and the total number of non-zeros of the sparse matrix, respectively.

- Real array $A[\,]$ of size of $nnz$ contains the non-zero entries row by row.
- Integer array $J[\,]$ of size of $nnz$ is made of the corresponding column indices for each non-zero entry in $A[\,]$.
- Integer array $I[\,]$ is made of the start pointer of each row in $A[\,]$ and $J[\,]$. The size of $I[\,]$ is $N+1$, $I[N+1]=nnz+1$. The number of non-zeros of the $i$th row can be expressed as $I[i+1]-I[i]$.

There are two basic ways to implement SpMV on GPU based on CSR format: CSR scalar (CSRS) and CSR vector (CSRV). CSRS calculates one row by one thread. Since the non-zero values and column indices are stored row by row in $A[\,]$ and $J[\,]$, so all threads access data in discontinuous way. This is why its performance is poor on GPUs.

The CSRV format is proposed in [5] and modified in [7] to realize the memory access contiguously. Unlike CSRS, it uses a half-warp (generally, 16 threads) to compute each row. All of the threads in a half-warp access the memory contiguously, so the chance of coalescing will be higher. Each thread of a half-warp compute partial result and stored in share memory and it's need to reduce the partial result to sum up when all calculation finished [8].

## 2.3. ELL-Like Formats

### 2.3.1. ELL Format

Ellpack format (ELL, in brief) is well suited to vector architectures [5]. For a $N \times M$ matrix with a maximum of $K$ non-zeros every row, the ELL format stores the non-zero elements in each row to the left side in a dense $N \times K$ array. The corresponding column indices are store in another two dimension array.

- Real two dimension array $A[\,]$ contain the non-zeros entries.
- Integer two dimension array $J[\,]$ is made of the column indices for each non-zero entry in $A[\,]$.

Each row of ELL format with the number of non-zeros less than $K$ needs to be padded with zeros. For ease of the calculation on GPUs, it is a common way to write a two dimension array as a one dimensional array, column by column. Then $A[i+j\times N]$ represents element of the $i$th row and the $j+1$ th ( $j=0,1,\cdots,K-1$ ) column.

ELL can be considered as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. When the numbers of non-zeros in each row are almost equal, the zeros need to be padded will be less, which leads to a high performance of the implementation of SpMV on GPUs. On the other hand, when difference of the number of non-zeros between rows is large, more zeros need to be padded, which will decrease the performance.

### 2.3.2. ELLR Format

ELLPACK-R format (ELLR, in brief) made some changes and optimizations on ELL format. It consists of three one dimension arrays, $A[\,]$, $J[\,]$, and $rl[\,]$.

- $A[\,]$ and $J[\,]$ are same as ELL format.
- Integer array $rl[\,]$ contains the numbers of non-zeros per row. The size of $rl[\,]$ is $N$ (*i.e.* the number of rows of the matrix).

These three arrays are represented in **Figure 2**. This format have some advantage, which can be seen in [6] for more detail. Some of its characters are listed in the following:

1) The coalesced global memory access, thanks to the column-major ordering used to store the matrix elements [6].

2) Non-synchronized execution between different blocks of threads.

3) The reduction of the waiting time or unbalance between threads of one warp [6]. **Figure 3** give an example of ELLR, which assume a warp consists 8 threads. The darker areas are the non-zero elements that need to be computed. The lighter areas are the waiting times. The number of iterations needed in a warp equals to the largest number of non-zeros in rows (e.g., the first warp needs 4 iterations in **Figure 3**). It reduces many iterations compare with ELL format.

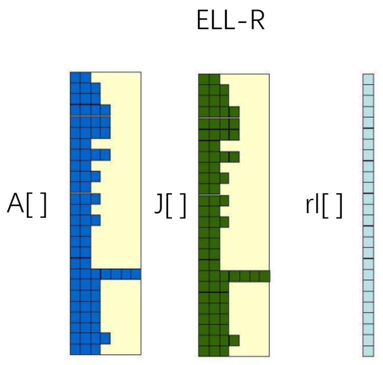

**Figure 2.** ELLR format.

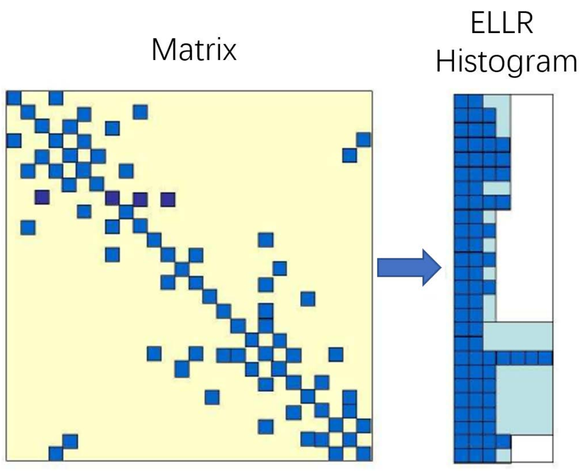

Figure 3. ELLR format for every warp (8 threads) [6].

4) Homogeneous computing within the threads in the warps.

### 2.3.3. BiELL Format

BiELL format is a bisection ELL format [8]. Based on ELL format, it sort the rows in a warp according to the numbers of non-zeros in each row, then group the rows in a warp by bisection technique. The detailed process can be seen in [8]. BiELL format consists of four arrays, $A[\,]$, $J[\,]$, $I[\,]$ and $perm[\,]$.

- Real array $A[\,]$ and integer array $J[\,]$ store the non-zeros and corresponding column indices column by column and warp by warp.
- Integer array $I[\,]$ contain the starting pointers of the first element in each group.
- Integer array $perm[\,]$ records the order of rows.

A simple example is given in **Figure 4**, which assume there is 4 threads in a warp [8].

The main advantage of the BiELL format is that it balances the workload of different threads in a warp, so reduces the waiting time. By using bisection technique, the non-zero elements in a group are equally allocated to different threads. This reduces the number of zeros to be padded and the number of iterations.

### 2.3.4. HYB Format

The hybrid format (HYB) is a combination of the ELL and COO formats. The purpose of the HYB is to store the non-zeros of a given number per row in the ELL data structure and the remaining entries in the COO format [5]. How to select the storage portion of the ELL is a difficult point in the HYB format. One way profitable to do this is to add the $K$th column to ELL structure if at least one third of the matrix rows contain $K$ (or more) non-zeros. The other way is to select the average number of non-zeros of each row to storage in ELL format.

### 2.3.5. JAD and BiJAD Format

The jagged diagonal (JAD) format [7] [14] sorts the rows based on the numbers

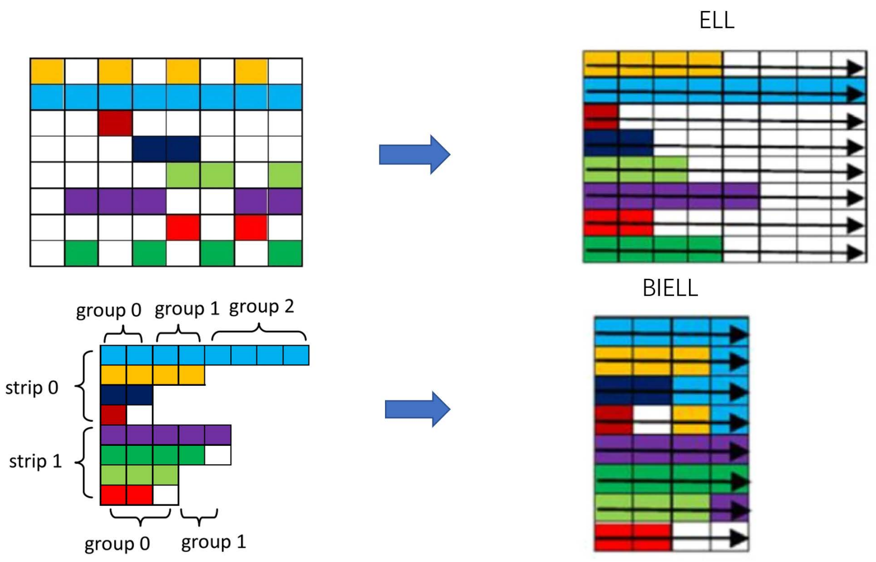

**Figure 4.** A simple example of BiELL.

of non-zeros of each row, then stored the non-zeros in jagged diagonals. It consists of four arrays, $A[\,]$, $J[\,]$, $I[\,]$ and $perm[\,]$.

- Real array $A[\,]$ and integer array $J[\,]$ store the non-zeros and its corresponding column indices jagged diagonal by jagged diagonal.
- Integer array $I[\,]$ contains the starting position of the first element in each jagged diagonals.
- Integer array $perm[\,]$ records the order of rows.

JAD reduces the number of zeros to be padded, which leads to a better performance than the ELL format.

The bisection JAD (BiJAD) format is a bisection of JAD, which is an optimized and improved version of JAD on GPUs. BiELL sorts each row in a warp, while BiJAD sorts all the rows. The BiJAD format may decrease the padding zeros compared with BiELL format; however, when the results are permuted back to the origin order, the pattern memory accessed may not be coalescent [8].

## 3. Our New Format: PELLR

In order to optimizing the SpMV on GPUs, we propose a new format, PELLR format. It is based on the permutation of row for ELLR format.

PELLR format sorts the rows based on the number of non-zeros of each row, then stored the non-zeros in ELL format. It consists of four one dimension arrays, $A[\,]$, $J[\,]$, $rl[\,]$ and $perm[\,]$.

- $A[\,]$, $J[\,]$ and $rl[\,]$ are same as ELLR format.
- Integer array $perm[\,]$ records the order of rows.

The size of $rl[\,]$ is $N$ (*i.e.* the number of rows of the matrix), which purposes to easy theory analysis. In the actual calculation, we can take the size of $rl[\,]$ as $N_w$ (much less than $N$, see in follows), which reduce the need of memory.

PELLR mainly optimizes the third character of ELLR format. For a sparse matrix of size $N \times M$, the difference of non-zeros of each row may be very large. GPU calculation is based on a warp as a whole. It's going to happen frequently that a row consist of little non-zeros (e.g., <5), while another row consist many non-zeros (e.g., >20) belong to a warp. This creates extra unnecessary computational workload. Our idea is to sort the whole row according to the number of non-zeros in each row, so the rows with more non-zero elements would be arranged together, and the rows with fewer elements would be grouped together. This will reduce unnecessary calculations and obtain an optimized version of storage format. An example is given in Figure 5.

Now, we give some analysis and compare of ELLR and PELLR. In order to better describe how to sum the total work amount, we use the following denotes:

- $nnz$ : the total number of non-zeros.
- $N$: the number of rows.
- $rl$ : an array for the number of non-zeros of each row.
- $warp$ : has 32 threads, $warp = 32$.
- $N_w$ : $\left\lfloor \frac{N+32-1}{warp} \right\rfloor$, the number of $warp$ for the matrix. $\lfloor\ \rfloor$ means to take an integer.
- $N_{iter}$ : the total number of iterations.
- $N_p$ : the number of computations.
- $b_i$ : an array of size of $warp$ contains the number of non-zero elements in each row in the $i$th warp. In the final warp, we take zero for the row that doesn't exist. We have the following relation

$$b_i = rl\left[warp \times (i-1)+1, \cdots, warp \times i\right],\ i = 1, \cdots, N_w.$$

Then we can deduce that the number of iterations and work amount are:

$$N_{iter} = \sum_{i=1}^{N_w} \max\left\{b_i[1], \cdots, b_i[32]\right\} \tag{1}$$

$$N_p = \sum_{i=1}^{N_w}\left(\sum_{j=1}^{warp} b_i[j] + \sum_{j=1}^{warp}\left(b_i[j]-1\right)\right) = \sum_{i=1}^{N_w}\sum_{j=1}^{warp}\left(2b_i[j]-1\right) \tag{2}$$

For a matrix, we can use these two expressions to obtain the amount of computation and the number of iterations. For the ELLR and PELLR formats, it can

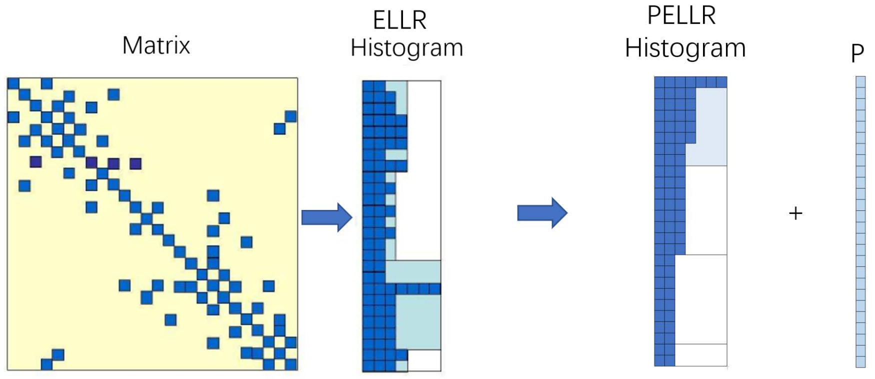

Figure 5. PELLR format (P is the array $perm[\,]$).

be judged from equations of Equation (1) and Equation (2) that the total number of iterations has changed, but the amount of calculation has not changed. For the matrix given in **Figure 5**, $N_w = 4$, we have

$$\begin{aligned} b_{1,ellr} &= [2,3,3,4,4,4,2,4], & b_{1,pellr} &= [7,4,4,4,4,4,3,3], \\ b_{2,ellr} &= [2,3,2,3,2,3,2,2], & b_{2,pellr} &= [3,3,3,3,3,3,3,3], \\ b_{3,ellr} &= [2,2,7,3,3,3,3,3], & b_{3,pellr} &= [2,2,2,2,2,2,2,2], \\ b_{4,ellr} &= [4,3,0,0,0,0,0,0], & b_{4,pellr} &= [2,2,0,0,0,0,0,0]. \end{aligned}$$

So

$$N_{iter,ellr} = 18, \quad N_{iter,pellr} = 14, \quad N_{p,ellr} = N_{p,pellr}.$$

We reduce the number of iterations by a new permutation of the rows. In this example, PELLR only needs 14 iterations, less than 18 iterations needed by ELLR.

## 4. Numerical Result

Our experiments are run on a personal computer equipped with NVIDIA Quadro P600; the operating system is a 64-bit Linux with CUDA 10.0 driver. The SDK and CUDA Toolkit, CUSPARSE [15], are used for programming. All the programs are based on CUDA-ITSOL [16] released by Li and Saad.

All the test matrices in our experiments are real square matrices collected from Matrix Market and the university of Florida sparse matrix collection. Information about the matrix is listed in **Table 1**. The parameters used in the table as follows:

$N$: the matrix row size.

$nnz$ : the non-elements number for matrix.

$ave$ : the average of non-zeros per row, $ave = nnz/N$ .

$\sigma$ : the standard deviation of the number of non-zeros elements per row. $\max - \min$ : is the difference between the maximum and minimum of non-zeros elements per row.

Performance in GFlops is calculated as $2 \times nnz/T$ , where $T$ is the wall time of SpMV calculated on the GPU. In order to improve the performance of SpMV, we used texture memory to store the vector $x$ for the SpMV kernels. This memory is bound to the global memory and plays the role of a cache level within the memory hierarchy [3].

In **Figure 6**, the experimental results of ELLR and PELLR formats are presented. The ordinate is the ratio of the performance of PELLR format against that of ELLR format. The abscissa is arranged based on $\sigma$ of the matrices (and the axes in the following figures are arranged in the same way). It can be seen that the performance of PELLR format is better than that of ELLR format for all matrices. The benefits are significant for matrices with large parameters of $\sigma(>10)$, which means that the difference of the number of non-zero between rows is large, such as matrix **bcsstk24**, **cavity25** and **e40r5000**. In this case, PELLR format is faster than ELLR format about 1.5 times. On the other hand,

Table 1. Test matrices.

| Matrix | $N$ | $nnz$ | $ave$ | $\sigma$ | max-min |
|---|---|---|---|---|---|
| rdb2048l | 2048 | 12,032 | 2.9 | 0.34 | 2 |
| dw2048 | 2048 | 10,114 | 4.9 | 0.51 | 5 |
| dw8192 | 8192 | 41,746 | 5.1 | 0.61 | 5 |
| qh1484 | 1484 | 6110 | 4.1 | 1.60 | 11 |
| mhd4800b | 4800 | 27,250 | 5.7 | 2.00 | 9 |
| s3dkt3m2 | 90,449 | 1,921,955 | 21.2 | 2.39 | 38 |
| s3dkq4m2 | 90,449 | 2,455,670 | 27.1 | 2.67 | 44 |
| sherman3 | 5005 | 20,033 | 4.0 | 2.70 | 6 |
| gemat12 | 4929 | 33,044 | 6.7 | 3.00 | 42 |
| Insp3937 | 3937 | 25,407 | 6.5 | 3.10 | 10 |
| mhd3200a | 3200 | 68,026 | 21.0 | 5.80 | 32 |
| utm5940 | 5940 | 83,842 | 14.0 | 6.30 | 29 |
| bcsstk24 | 3562 | 159,910 | 45.0 | 11.00 | 42 |
| msc23052 | 23,052 | 1,154,814 | 50.1 | 11.60 | 166 |
| bcsstk36 | 23,052 | 1,143,140 | 49.6 | 12.20 | 170 |
| e20r4000 | 4241 | 131,430 | 31.0 | 15.00 | 54 |
| e40r5000 | 17,281 | 553,562 | 32.0 | 16.00 | 54 |
| cavity25 | 4562 | 131,735 | 29.0 | 17.00 | 61 |
| boneSo1 | 127,224 | 6,715,152 | 52.8 | 17.64 | 69 |
| memplus | 17,758 | 126,150 | 7.1 | 22.00 | 572 |

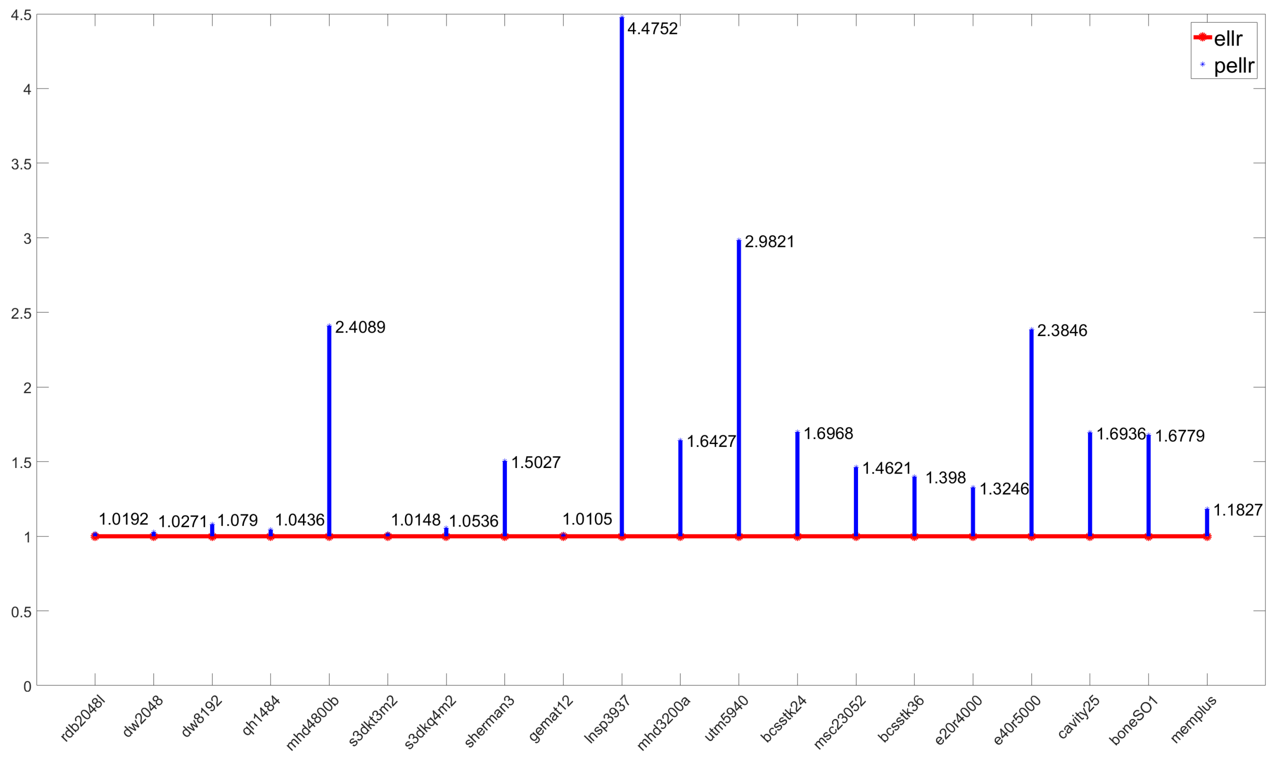

Figure 6. The performance comparison of ELLR and PELLR.

for matrices with small $\sigma(<3)$, PELLR format has no obvious advantage to ELLR format, such as the matrix **dw2048**, **qh1484** and **s3dkt3m2**.

The matrices **memplus** and **lnsp3937** are special. The structure of **memplus** is shown in Figure 7, which the dots in different colour represent non-zero elements. It has a $\sigma$ of 22, but the ratio of performance is only has 1.1. Another fact for it is $avg = 7.1$ is small and $\max - \min = 572$ is really large, which means only few rows has many non-zeros entries in the matrix. The Figure 7 also proves this view of point. We can observe that there are many dots focus on the diagonals, the upper and the left sides of the matrix. So this matrix has only few rows in the front which have a lot of non-zero elements. So advantages of permutate are not distinct. The matrix **lnsp3937** has $\sigma = 3.1$ and $\max - \min = 10$, but the ratio of performance reaches 4.5. This is because the rows which have almost equal length are scattered, permutation groups the together and the number of iterations is reduced dramatically.

From a large amount of experiments, we can make a general remark that PELLR format is faster than ELLR for almost all matrices, and when the matrix has the parameters of $\sigma > 10$, $ave > 20$ and $\max - \min > 10$, PELLR format is faster than ELLR format about a factor of 1.5.

We have compared the performance PELLR format with HYB format, the results are give in Figure 8. The program for HYB format we used is from CUSPARSE library [15]. We see again that the PELLR format has a significant advantage over HYB format. But in this case, the trend of ratio is different. With the increase of $\sigma$, the advantage of PELLR format trends to decrease. For matrices with small $\sigma < 10$, the average factor of speedup is approximately 4. This is because HYB format use a factor of 1/3 (default in CUSPARSE) to separate the ELL and COO. What the defect of HYB in this case is comes from COO. While for large $\sigma > 10$, the ratio is approximately 1.6, which shows that the effect of COO in this case is not so apparent.

Some matrices have special results due to their structures. **msc23052** and **bcsstk36** have large $\max - \min$ (166 and 170, respectively), the ratio is only 1.2. **memplus** (see Figure 7) has $\max - \min = 572$, the ratio of 0.7 shows that PELLR is slower than HYB and the advantage of permutation is overwhelmed.

In Figure 9, we present the experimental results for PELLR and BiELL formats. The PELLR format still has advantages for most cases (17 vs. 3). It is worth

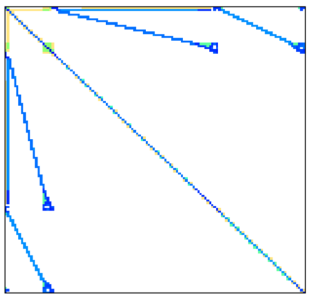
Figure 7. The matrix structure of memplus.

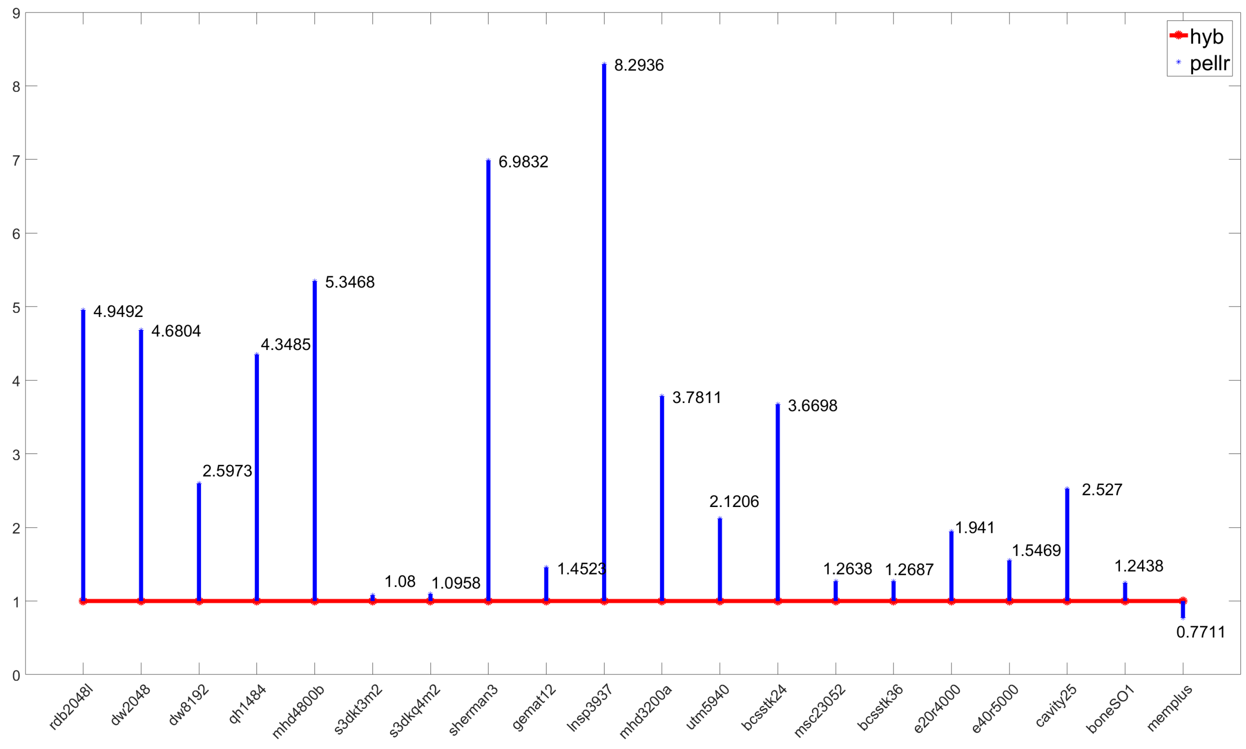

**Figure 8.** The performance of different formats: PELLR and HYB.

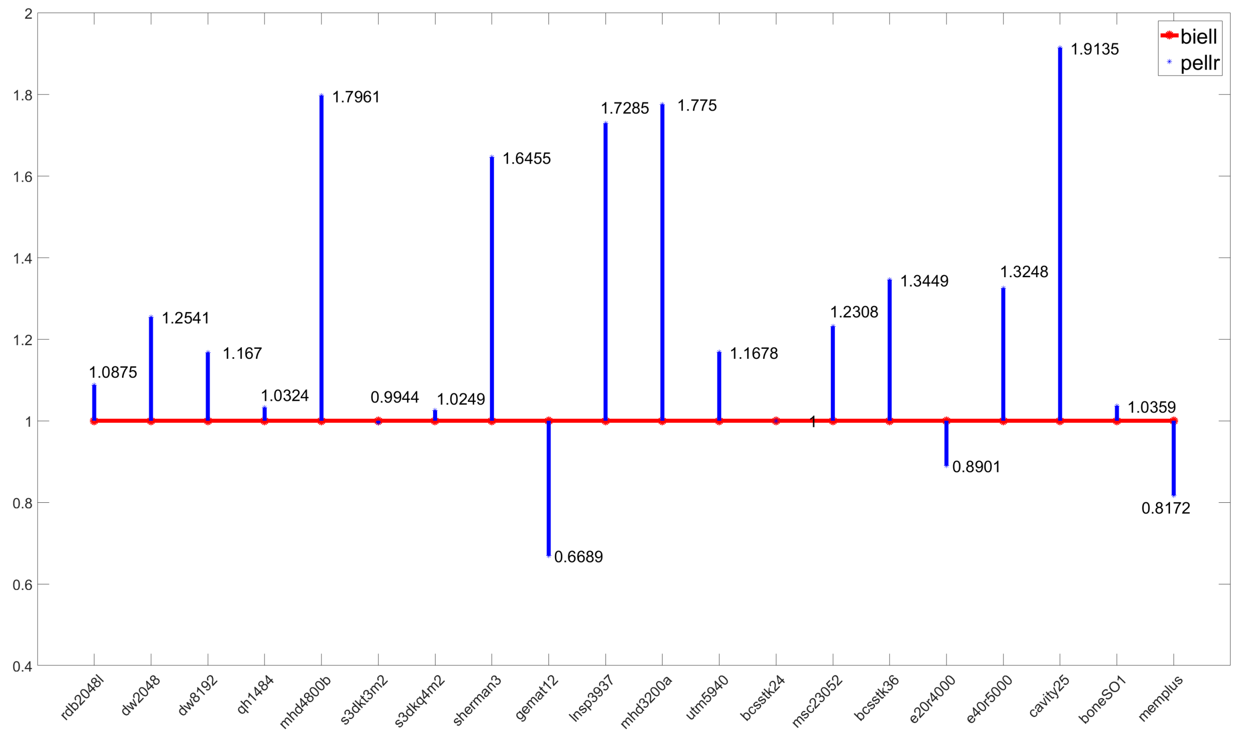

**Figure 9.** The performance of different formats. PELLR and BiELL.

to note matrices **lnsp3937** and **mhd4800b**. Their $\sigma$ and $\max-\min$ are 3.1 and 2, and 10 and 9, respectively. Since they are relatively small, the number of iterations per warp for BiELL format will not reduce very much, and the time of the judgment statement (in SpMV kernel on GPU) is not covered, so PELLR has obvious advantage in this cases. For matrix **memplus**, which structure is seen in Figure 7, the first warp of BiELL will reduce many iterations, so the ratio is 0.8172.

In Figure 10, PELLR and BiJAD are compared. Both formats sort all the rows in a whole. So the performances of two formats are nearest. The difference is BiJAD divides each warp into six groups to reduces the number of iterations, while PELLR format only relies on $rl[\,]$. So it is important that how many iterations are reduced in the warp in BiJAD SpMV kernel. If the number of iterations reduced is less, the effect of the judgment statement may be not covered and the benefits obtained may be less. Therefore, for these matrices, the experimental results depend on the characteristics of each matrix. For our 20 test matrices, PELLR format is advantageous in 15 cases (75%), while the highest ratio is only 1.3.

Figure 11 shows a comparison of PELLR, HYB, BiELL, and BiJAD formats. We found that the PELLR, BiELL, and BiJAD formats are better than the HYB format in most cases, especially when parameter $\sigma$ is relatively small (<10). For our test matrices, we find the performance of HYB format is superior to that of other formats only in the case of **memplus**.

Figure 12 gives the comparison of more formats, which are CSRV, JAD, ELL, ELLR, PELLR, HYB and cuCSR. The cuCSR format is CSR format provided by CUSPARSE. What we can see from Figure 12 is listed as follows:

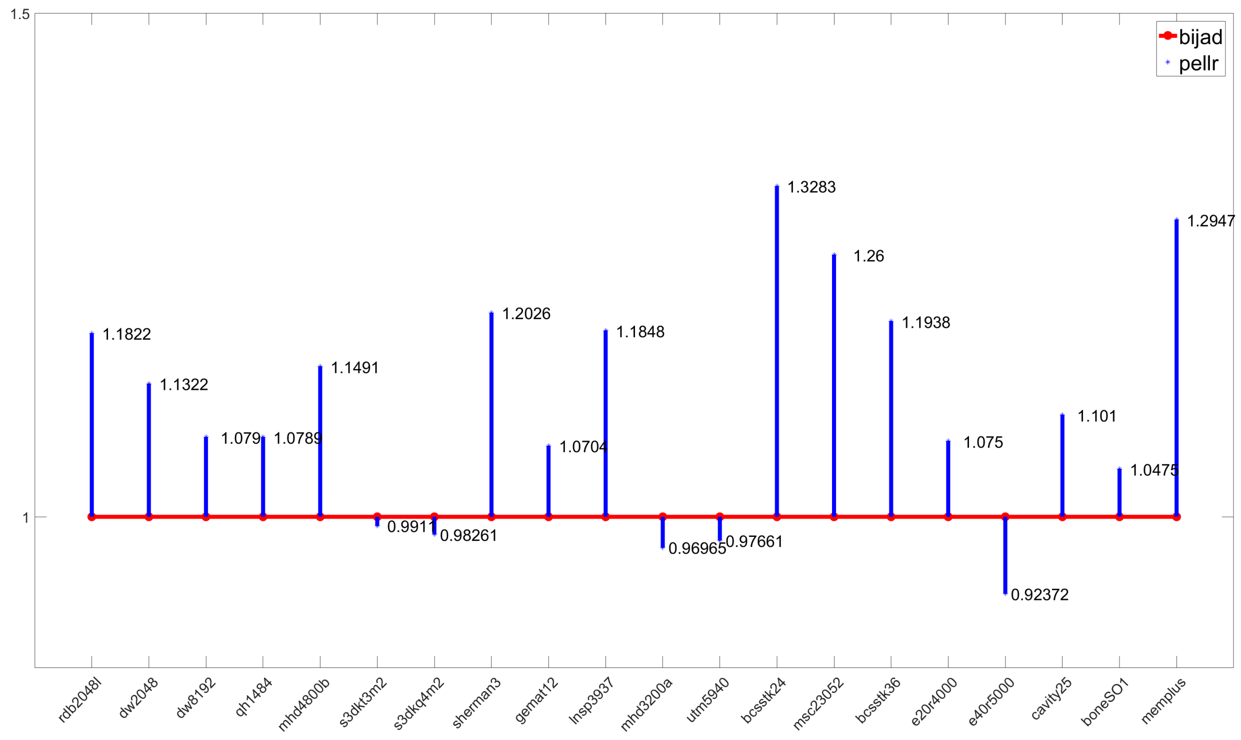

Figure 10. The performance of different formats: PELLR and BiJAD.

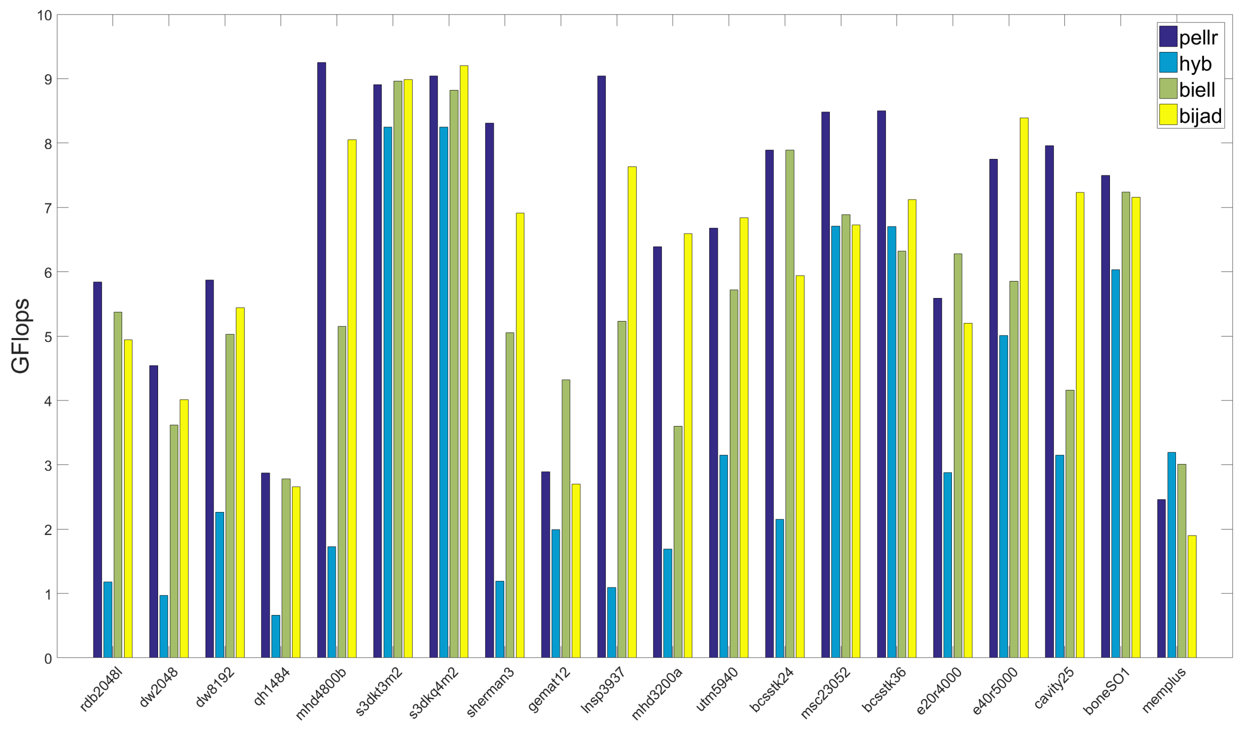

**Figure 11.** The performance of PELLR, HYB, BiELL, BiJAD formats.

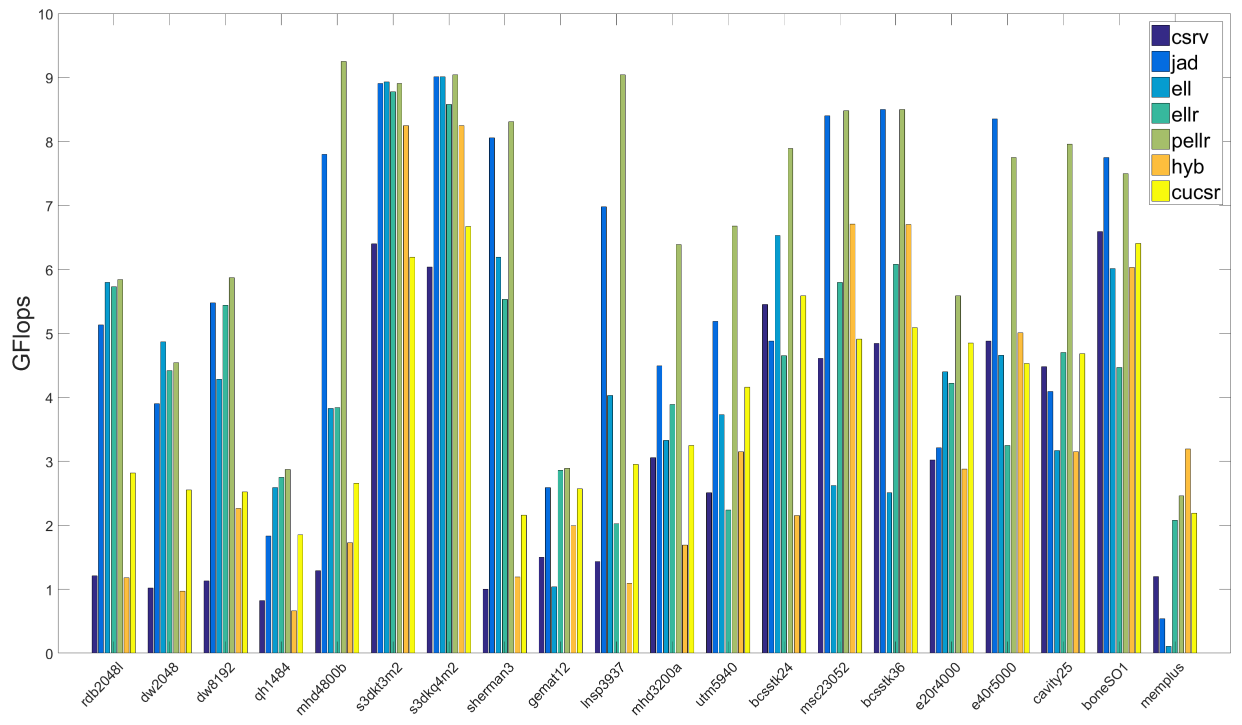

**Figure 12.** The performance of different formats. For every matrix, formats listed from left to right are CSRV, JAD, ELL, ELLR, PELLR, HYB, cuCSR.

1) Overall, the PELLR format is optimal in most cases. And then the JAD and ELLR formats also turned out pretty well, the CSRV format is relatively poor.

2) The performance of CSRV and cuCSR is sensitive to the $ave$. In general, CSRV is poorer than cuCSR in most cases. CSRV will performance well when $ave$ is large (>16), such as **boneSo1** and **bcsstk24**.

3) HYB is not as good as we thought for our test matrices. But it can be found good performance when $ave$ and $\sigma$ are bigger, such as **bcsstk36** and **msc23052**, and its performance is best for matrix **memplus**.

4) For almost matrices, PELLR and JAD are most outstanding, which in turn they are the best case.

5) All statements in the previous experiments is obtained again.

## 5. Conclusions

We proposed a permutated ELLR format by sorting the rows based on the number of non-zeros (or the length) of each row. This preprocessing makes the rows of almost equal length together. So the number of the iterations is reduced and the performance of SpMV can be improved, the speed up achieved about 1.5 in our experimental results. Furthermore, we deduced the formulation of the number of iterations and the work amount, which can be used to evaluate the performance of SpMV. In our experiments results, the performance of PELLR format is best in most cases. The performance comparison of different matrix formats is given, and some special cases are explained.

The PELLR format improves ELLR format in performance of SpMV, but it also adds increased storage memory. This format stores two more arrays than ELL format, $rl[\,]$ and $perm[\,]$. In the future, we want to reduce the memory of PELLR format and how to choose an optimal storage format for A sparse matrix by the matrix parameters and Equation (1).

## Acknowledgements

This work was supported in part by Science Challenge Project, No. TZ2016002 and NSF of China (61472462, 11671049, 11601033).

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Hong, S. and Kim, H. (2009) An Analytical Model For a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness. 36*th International Symposium on Computer Architecture* (*ISCA* 2009), Austin, TX, 20-24 June 2009, 152-163. https://doi.org/10.1145/1555754.1555775

[2] Kothapalli, K., Mukherjee, R., Rehman, M.S., Patidar, S., Narayanan, P.J. and Srinathan, K. (2009) A Performance Prediction Model for the CUDA GPGPU Platform. *Proceedings of* 16*th International Conference on High Performance Computing, HiPC* 2009, Kochi, India, 16-19 December 2009, 463-472. https://doi.org/10.1109/HIPC.2009.5433179

[3] NVIDIA (2018) NVIDIA CUDA C Programming Guide. Version 10.0.

[4] NVIDIA, CUBLAS (2019) https://developer.nvidia.com/cublas/

[5] Bell, N. and Garland, M. (2008) Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation.

[6] Vázquez, F., Garzón, M., Martínez, J.A. and Fernández, J.J. (2009) Accelerating Sparse Matrix Vector Product with GPUs. *Proceedings of the* 2009 *International Conference on Computational and Mathematical Methods in Science and Engineering*, CMMSE 2009, 30 June, 1-3 July 2009, 1081-1092.

[7] Li, R. and Saad, Y. (2012) GPU-Accelerated Preconditioned Iterative Linear Solvers. *The Journal of Supercomputing*, **63**, 443-466. https://doi.org/10.1007/s11227-012-0825-3

[8] Zheng, C., Gu, S., Gu, T.-X. and Liu, X.-P. (2014) BiEll: A Bisection ELLPACK Based Storage Format for Optimizing SpMV on GPUs. *Journal of Parallel and Distributed Computing*, **74**, 2639-2647. https://doi.org/10.1016/j.jpdc.2014.03.002

[9] Baskaran, M.M. and Bordawekar, R. (2009) Optimizing Sparse Matrix-Vector Multiplication on GPUs. IBM Research Report, RC24704 (W0812-047), IBM Corporation.

[10] Langr, D. and Tvrdik, P. (2016) Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems*, **27**, 428-440. https://doi.org/10.1109/TPDS.2015.2401575

[11] Liu, W.F. and Vinter, B. (2015) CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. *ACM on International Conference on Supercomputing-ICS*, Newport Beach, CA, 8-11 June 2015, 339-350. https://doi.org/10.1145/2751205.2751209

[12] Vázquez, F., Ortega, G. and Fernández, J.J. (2010) Improving the Performance of the Sparse Matrix Vector Product with GPUs. 10*th IEEE International Conference on Computer and Information Technology*, *CIT* 2010, Bradford, West Yorkshire, UK, 29 June-1 July 2010, 1146-1151. https://doi.org/10.1109/CIT.2010.208

[13] Saad, Y. (2003) Iterative Methods for Sparse Linear Systems. 2th Edition, Society for Industrial Applied Mathematics, New York. https://doi.org/10.1137/1.9780898718003

[14] Cenvahir, A., Nukada, A. and Matsuoka, S. (2009) Fast Conjugate Gradients with Multiple GPUs. *Computational Science-ICCS* 2009, 9*th International Conference*, Baton Rouge, LA, 25-27 May 2009, 893-903. https://doi.org/10.1007/978-3-642-01970-8_90

[15] NVIDIA, CUSPARSE (2019) https://developer.nvidia.com/cusparse/

[16] Li, R. and Saad, Y. (2011) CUDA-ITSOL. http://www-users.cs.umn.edu/~saad/software/CUDA_ITSOL/