Proceedings of ALGORITMY 2012
pp. 282–290

IMPROVED ROW-GROUPED CSR FORMAT FOR STORING OF SPARSE MATRICES ON GPU *

TOMÁŠ OBERHUBER † AND MARTIN HELLER †

Abstract. We present new format for storing sparse matrices on GPU. We compare it with several other formats including CUSPARSE which is today probably the best choice for processing of sparse matrices on GPU in CUDA. Contrary to CUSPARSE which works with common CSR format, our new format requires conversion. However, multiplication of sparse-matrix and vector is significantly faster for many matrices. We demonstrate it on set of 1 600 matrices and we show for what types of matrices our format is profitable.

Key words. sparse matrices, SpMV, parallel computing, GPU, thread computing, CUDA, circuit simulations, optimizations problems

AMS subject classifications. 65F50, 65F10, 65Y05

1. Introduction.

GPUs are specialised devices offering high computational power as well as high memory throughput [7]. Their architecture is very similar to processor arrays [14]. GPUs are efficient for algorithms exhibiting massively parallel and homogenoues computations. Eventhough most operations from linear algebra are not computationaly intensive, corresponding algorithms can profit from high memory throughput reaching almost 200 GB/s. This has been demonstrated in many works dealing with dense matrices [9, 6, 16]. Situation is more complicated in case of sparse matrices. They often have irregular pattern of non-zero elements and they significantly reduce available parallelism. It makes processing of the sparse matrices on GPU difficult but also challenging.

In this article we concentrate on the sparse-matrix vector multiplication (SpMV) since it is a key part of many iterative solvers for linear systems. The performance is influenced mainly by the format used to store the matrix. Formats for storing the sparse matrices often involves additional data, typicaly column indexis. On the vector architectures, the data must be aligned in the memory. CUDA developers speak about coalesced global memory accesses. Their importance for the efficiency of SpMV is discussed in [2, 11]. For this reason, artificial zeros must be often inserted. Moreover, with different number of non-zero elements in each row, the multiprocessors may be load unequaly. Efficient format should address the following:

1. reduce amount of additional data while keep coalesced global memory accesses,
2. distribute the non-zero elements of the matrix evenly to multiprocessors

*This work was partially supported by the Jindřich Nečas Center for Mathematical Modelling, Research center of the Ministry of Education of the Czech Republic LC06052, Research Direction Project of the Ministry of Education of the Czech Republic No. MSM6840770010, and Advanced Supercomputing Methods for Implementation of Mathematical Models, project of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14, 2011-13.

† Department of mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, Praha 2, 120 00, Czech Republic tomas.oberhuber@fjfi.cvut.cz

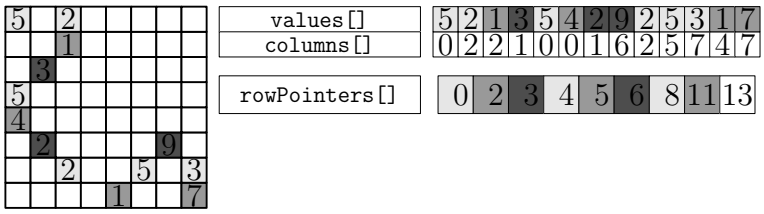

FIG. 2.1. *CSR format*

1.1. Contribution. We present new format for storing the sparse-matrices on GPU. It is optimised for matrix-vector multiplication. With some types of matrices, it is several times faster compared to today state-of-the-art formats like CUSPARSE, Hybrid [2] (combination of ELLPACK and COO format – it is implemented in CUSP library [4]), Row-grouped CSR [11] and Sliced ELLPACK [8]. We analyse on what type of matrices the new format outperforms the others and vice versa.

1.2. Organisation. The article is organised as follows. In Section 2, we briefly summarise existing formats for storing sparse matrices on GPU. The new format is presented in Section 3 together with description of conversion from CSR format. The implementation in CUDA with a source code of matrix-vector multiplication kernel is in Section 4. Achieved results with short performance analysis are topics of the last Section 5.

2. Formats for sparse matrices on GPU. CSR (Compressed Sparse Rows) format (Figure 2.1) is standard and most popular format for storing sparse matrices [15]. It consists of arrays `values` storing rowisely the non-zero elements of the matrix, `columns` storing column index of each non-zero element and `rowPointers` containing offset of each matrix row in arrays `values` and `columns`. This format is easy to implement. Storing data in arrays improves efficiency of data transfer.

Works by Bell and Garland [2] or Bautois et al. [3] showed that this format is inefficient on GPU for matrix-vector multiplication. Better formats are based on the ELLPACK format (Figure 2.2) by Monakov and Avetisian [8] or our similar format studied in [11, 13]. Formats based on ELLPACK require homogeneous distribution of non-zero elements in rows. If the number of non-zero elements in each row is very different, the ELLPACK format loses efficiency. Bell and Garland [2] proposed Hybrid format which is part of CUSP library. It has also achieved great popularity. Recently CUSPARSE [10] showed that even pure CSR format can be implemented efficiently on GPU. Nevertheless, tests on large sets of sparse matrices like those in [11] show that there are a lot of matrices for which common CPU performs better. There is still great potential to improve formats for sparse matrices on GPU. The ELLPACK format, as depicted on the Figure 2.2, allocates the same number of elements for each matrix row. If there are less non-zero elements on some row, artificial zeros are added to align the data. This increases memory requirements and slow down matrix-vector multiplication because more data must be transfered. On GPU, usualy one thread is mapped to one row. Arrays `values` and `columns` are stored in global memory and so the accesses to these arrays must coalesced (see [2, 11]). This is reason why these arrays are stored columnwise instead of rowise. Since the coalesced memory accesses must be fulfilled only for threads in one warp we may split the matrix into slices of rows processed by the same warp. The slices are stored separately. If there

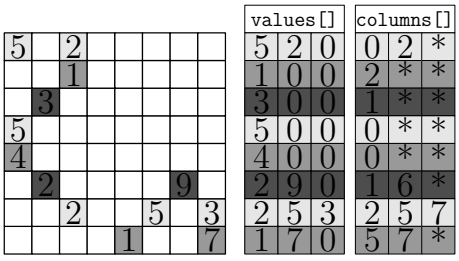

FIG. 2.2. *ELLPACK format*

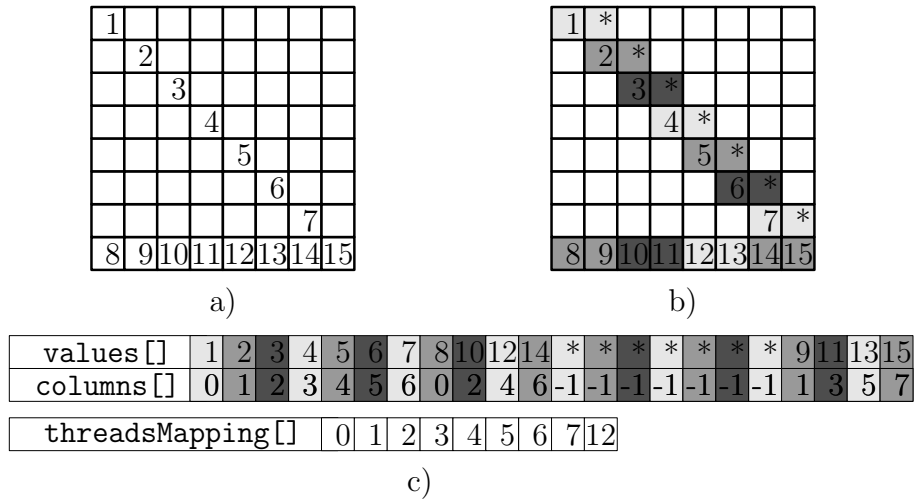

FIG. 3.1. *Example demonstrating the Improved Row-grouped CSR format: a) the original matrix, b) mapping of chunks to matrix rows – each chunk is depicted in different greyscale, c) arrays with the non-zero elements (*`values`*), column indecis (*`columns`*) and mapping of threads to rows (*`threadsMapping`*).*

is a row having sigificantly more non-zero elements than the others, only one slice is affected. This modification, introduced independentaly in [8, 11], reduces number of artificial zeros required by original ELLPACK format. However, for some matrices these formats still generate too many artificial zeros and matrix-vector multiplication can be several orders slower compared to CSR format on CPU.

The next step is to split long rows (i.e. with a lot of non-zero elements) and process them by more threads. The matrix is again divided into groups of rows. We introduce *chunks* of non-zero elements. Each chunk in the same group has the same size. One row can be splitted into more chunks (but one chunk cannot cross boundary of one row). Chunks are stored in the same way as matrix rows in the Ellpack format (also stored columnwise to get the coalesced memory accesses). When performing the matrix-vector multiplication, we map one thread to each chunk. First the chunks are processed which results to array of partial sums. If some row consits of more chunks, its partial sums must be summed to get the final result. In the following text, we explain this format in more details.

3. New Improved Row-grouped CSR format. Figure 3.1 shows an example of matrix for which the ELLPACK format is inefficient. All rows except the last one have only one non-zero element and the last row is full. Assume that we map 11 threads to this matrix. We first assign one thread to each row. The thread mapped to the last row must process 8 elements. Therefore the chunk size is 8 and all other threads must process 8 elements as well because they belong to the same warp. We would have to allocate 49 artificial zeros to align the data. Since we have 4 threads left we may use them to diminish the chunk size by mapping them to the last row. As depicted in the Figure 3.1, if there are 4 threads assigned to the last row, the chunk size is only 2 and we need to allocate only 7 artificial zeros. The number of threads 11 was chosen to fit well to our example. If there were 12 threads we would map 5 threads to the last row. It would not decrease the chunk size but we would have to allocate one more chunk with artificial zeros. The number of the artificial zeros would be 9. To ensure the coalesced global memory accesses we store the chunks in the same way as rows are stored in Sliced ELLPACK format [8] or Row-grouped CSR format [11]. In other words, if chunks were rows of some matrix, the matrix would be stored columnwise.

Large matrices are splitted into small groups of rows. Each group is defined by the following structure (Listing 1):

LISTING 1
Structure defining group of rows

```
1 struct irgcsrGroupInfo
2 {
3    int firstRow;
4    int size;
5    int offset;
6    int chunkSize;
7 };
```

Parameter `firstRow` is the first matrix row and `size` is number of rows in the group i.e. the group contains rows indexed from `firstRow` to `firstRow + size - 1`. Parameter `offset` points to the first element of the group in arrays `values` and `columns` and `chunkSize` describes number of elements in chunk. It is constant in each group but it can be different for different groups. The number of chunks is the same for each group and equals CUDA block size. For each group there are `chunkSize * blockDim.x` elements in arrays `values` and `columns` starting at position `offset`.

Let us now comment a conversion from the CSR format. For good performance, it is necesary that all groups have approximately the same number of non-zero elements. Otherwise the load balance of multiprocessors would be unequal. One parameter, entering the converting algorithm is `desiredChunkSize`. We allocate new group, read the matrix row by row and compute number of non-zero elements. Once it is larger than `desiredChunkSize * blockDim.x` or there would be more rows than `blockDim.x` in the current group, we close it and allocate new one. The groups are defined by array of structures `irgcsrGroupInfo`. In the next step of the conversion, we compute the chunk size in each group. This can be done in parallel. We start by mapping one thread to each row of the group. The chunk size would be now equal to the number of non-zero elements in row with greatest filling. We define *the chunk filling* as number of non-zero elements in given chunk. If there are some threads left, we always find row with greatest chunk filling (like the last row in Figure 3.1) and map one more thread to it. When all threads are distributed to the rows we can

compute the final chunk size. We also need to store mapping of threads to rows. For this we allocate array `globalThreadsMapping`. It contains number of threads mapped to each row of matrix. Then we perform exclusive segmented prefix-sum [17] where segments correspond to groups. As a result, indexes of threads mapped to a row number `rowInGroup` in a group with number `groupNumber` are given by numbers

`globalThreadsMapping[groupInfo[groupNumber]->firstRow+rowInGroup]`,
`globalThreadsMapping[groupInfo[groupNumber]->firstRow+rowInGroup+1]-1`.

The last step is filling of the arrays `values` and `columns` by data belonging to particular chunks. Since we know the chunk size in each group we know how many elements will be inserted by each group. This phase thus can be done in parallel as well. We read the data chunkwise from CSR format and copy them in appropriate order to the mentioned arrays.

In the next section we explain matrix-vector multiplication.

4. Implementation in CUDA. For better understanding, we show the source code of the kernel in CUDA – see Listing 2.

LISTING 2
Kernel for matrix-vector multiplication

```
1  template< class Real >
2  __global__ void spmvKernel(
3     Real* target,
4     Real* vect,
5     Real* values,
6     int* columns,
7     irgcsrGroupInfo* globalGroupInfo,
8     int* globalThreadsMapping )
9  {
10    extern __shared__ int sdata[];
11    const int* globalGroupInfoPointer =
12       ( const int* ) globalGroupInfo;
13    irgcsrGroupInfo* groupInfo =
14       ( irgcsrGroupInfo* ) &sdata[ 0 ];
15    int* threadsMapping =
16       ( int* ) &sdata[ 4 ];
17    Real* partialSums =
18       ( Real* ) &sdata[ 4 + blockDim.x];
19
20    int bId = blockIdx.x;
21
22    /****
23     * Fetch the group info from the global memory
24     */
25    if( threadIdx.x < 4 )
26       sdata[ threadIdx.x ] =
27          globalGroupInfoPointer[ 4 * bId + threadIdx.x ];
28    __syncthreads();
29
30    /****
31     * Fetch mapping of threads to rows.
32     */
33    if( threadIdx.x < groupInfo -> size )
34       threadsMapping[ threadIdx.x ] =
35          globalThreadsMapping[ groupInfo -> firstRow + threadIdx.x ];
36
37    /****
38     * Each thread computes partial sum in its chunk
39     */
```

```
40      Real sum = 0;
41      int threadOffset = groupInfo -> offset + threadIdx. x;
42      for( int i = 0; i < groupInfo -> chunkSize; i ++ )
43      {
44         const int column = columns[ threadOffset ];
45         if( column != -1 )
46            sum += values[ threadOffset ] * vect[ column ];
47         else
48            break;
49         threadOffset += blockDim. x;
50      }
51      partialSums[ threadIdx. x ] = sum;
52      __syncthreads();
53
54
55      /****
56       * Sum the partial sums in each row
57       */
58      if( threadIdx. x < groupInfo -> size )
59      {
60         sum = 0;
61         int begin( 0 );
62         const int row = groupInfo -> firstRow + threadIdx. x;
63         if( threadIdx. x > 0 )
64            begin = threadsMapping[ threadIdx. x - 1];
65         int end = threadsMapping[ threadIdx. x ];
66         for( int i = begin; i < end; i++ )
67            sum += partialSums[ i ];
68         target[ row ] = sum;
69      }
70   }
```

The kernel first fetch data with reuse to fast shared memory. We allocate shared memory (lines 10–18) for one `irgcsrGroupInfo` structure, array `threadsMapping` keeping track of thread indecis mapped to rows of the group and array `partialSums` meaning of which is explained later. Then we fetch the group info structure. To achieve coalesced memory access we employ four threads to this task (lines 25–28). Thread synchronisation is important here. Next, we may fecth array with mapping of threads to rows. The next part is independent on this array and therefore synchronisation is not necessary. Each thread takes its own chunk and perform multiplication of this part of matrix data with given part of input vector. Result of it is partial sum. We remind that we mark artificial zeros by column index `-1`. Once a thread reaches this column index it exits the loop on lines 42–50. If it happens in whole warp, it exits too and it ommits the rest of artificial zeros. The last step is summing of the partial sums. There is data dependency with the previous part and so thread synchronisation on the line 52 is necessary.

5. Results. The results we present in this section were obtained on a system equiped with CPU AMD Phenom II X6 1100T with 16 GB DDR3 RAM and GPU Nvidia Tesla C2070 having 6 GB GDDR5 with memory bandwidth 144 GB/s and ECC turned off. All tests were done in double precision and sequentialy on CPU. Testing matrices were fetched from matrix databases [5, 1]. The testing set contained almost 1 600 sparse square matrices.

The best performance was achieved with 128 threads in block. Test show that the parameter `desiredChunkSize` can have strong impact on the efficiency. Simple rule might be: the more regular the marix is (in sense that there are almost the same

non-zero elements in each row), the larger the desired chunk size should be. With desired chunk size 32 we have achieved the highest performance almost 18 Gflops with matrices `Schenk_AFE` originating in structural problems. With `desiredChunkSize` set to one, the performance droped to 11 Gflops. On the other, with the matrix `rajat23`, the performance was six times higher with `desiredChunkSize` 1 (5.1 Gflops and speed-up 11 compared to CSR on CPU) than with `desiredChunkSize` 32 (0.81 Gflops). In the rest of this section we present tests obtained with the desired chunk size set to 1 because it seems to be more general setting.

Figure 5.1 shows speed-up of tested formats compared to CSR format on CPU. The vertical axis shows the speed-up in logarithmic scale. Each curve is drawn by its own ordering of matrix index and height of curve at certain x-position cannot be compared. There is one curve for each format and the slower it decrease the better the format is. Our test shows that the Hybrid format, Row-grouped CSR format, CUSPARSE library and the new format are is faster for 726, 907, 994 and 1168 matrices of 1600 respectively. This figure also shows that there are few matrices for which CPU is two orders of magnitude faster. They are mainly small matrices having tens or hundreds of uknowns as our previous test in [11] show.

The next Figure 5.2 shows speed-up of the new format compared to the others. The vertical axis shows speed-up in logarithmic scale. As well as on Figure 5.1, each curve has its own ordering. The higher the curve is, the better the new format is. Our test show that the the Hybrid format is slower on 1318 matrices, Row-grouped CSR on 1072 matrices and CUSPARSE on 1358 matrices.

Both Figures 5.1 and 5.2 demonstrate that the performance of sparse-matrix and vector multiplication on GPU is very variable. If high performance is the top priority, one should test more formats and choose the best one. For example the CUSPARSE library is almost 4 times faster than the new format on `TSOPF` and `case39` matrix sets from [5]. Both types of matrices model "Transient stability-constrained optimal power flow" and usualy the Hybrid format outperforms the others in these cases. On the other hand, the new format is ten times faster than CUSPARSE (and at the same time 5-8 times faster than CSR on CPU) for matrices `raj`, `rajat`, `GHS_indef`, `IBM_EDA`. These matrices come from circuit simulations or optimizations problems. Original Row-grouped CSR format (or similar Sliced Ellpack) is almost twice faster for `norris/torso2` and `t2d_q` matrices originating in finite difference methods. The mentioned matrices are all difficult to visualise in this paper because of very large dimensions and we refer reader to [5] for more details.

Acknowledgement. The source code of the format is freely available as a part of the Template Numerical Library (TNL) at

http://geraldine.fjfi.cvut.cz/~oberhuber/doku-wiki-tnl.

This work was partially supported by the Jindřich Nečas Center for Mathematical Modelling, Research center of the Ministry of Education of the Czech Republic LC06052, Research Direction Project of the Ministry of Education of the Czech Republic No. MSM6840770010, and Advanced Supercomputing Methods for Implementation of Mathematical Models, project of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14, 2011-13.

REFERENCES

[1] Z. Bai, D. Day, J. Demmel, and J. Dongarra. Test matrix collection (non-hermitian

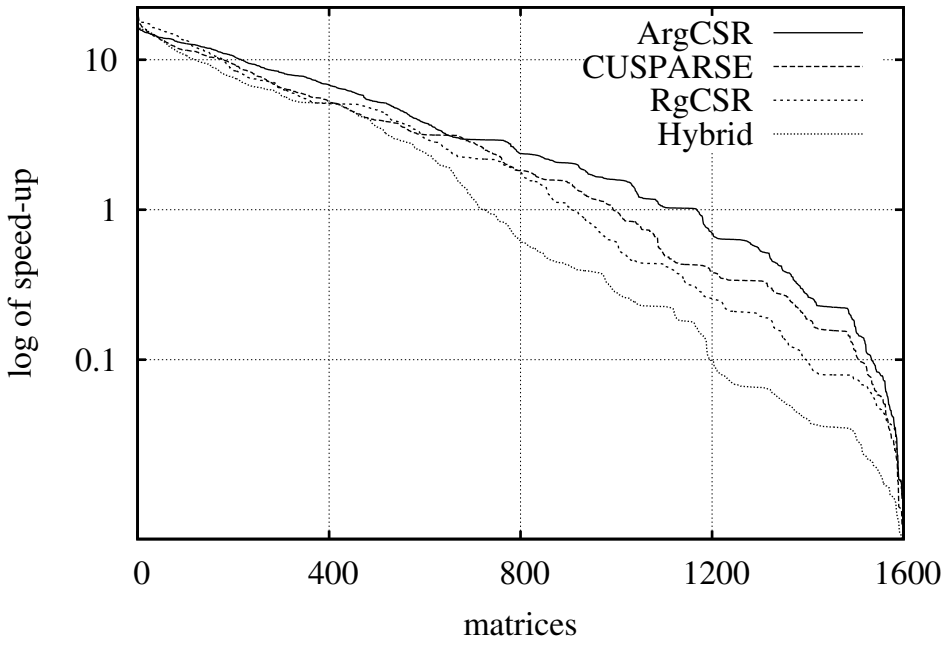

FIG. 5.1. *CSR comparison*

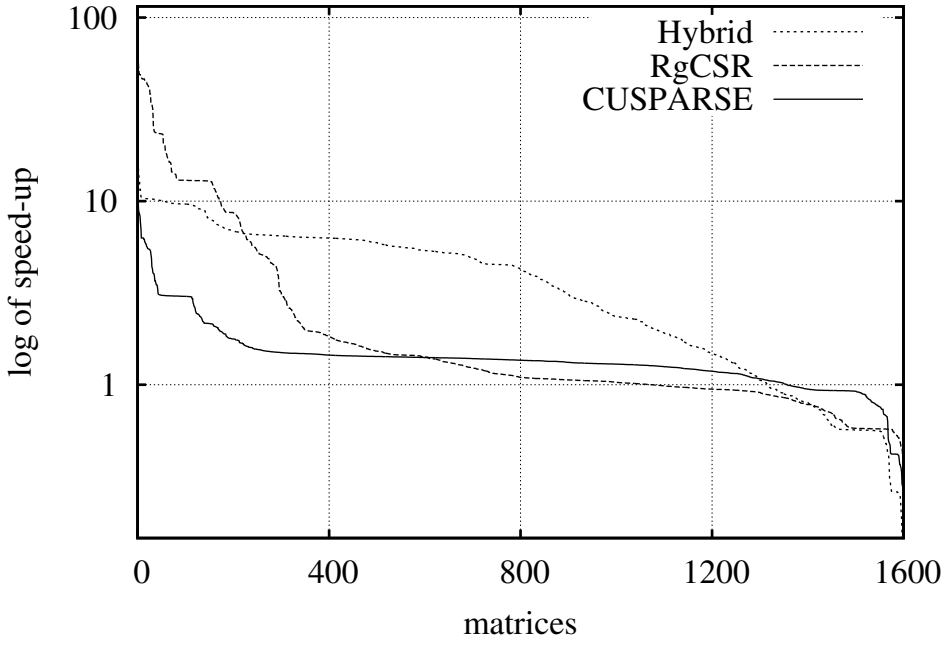

FIG. 5.2. *IrgCSR comparison*

eigenvalue problems), release 1. Technical report, University of Kentucky, 1996. ftp://ftp.ms.uky.edu/pub/misc/bai/Collection.

[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on cuda. Technical Report Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[3] L. Buatois, G. Caumon, and B. Levy. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009.

[4] *CUSP library* http://code.google.com/p/cusp-library

[5] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *NA Digest*, 92(42), 1994. http://www.cise.ufl.edu/research/sparse/matrices/.

[6] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware , page 3, washington, dc. In *2005 ACM/IEEE conference on Supercomputing – SC2005*, pages 3–14, 2005.

[7] V. W. Lee, Ch. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *ISCA'10*, pages 451–460, 2010.

[8] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures.

[9] Nvidia. *CUBLAS library*. Nvidia, 2010.

[10] Nvidia. *CUSPARSE Library User Guide*, 2012.

[11] T. Oberhuber, A. Suzuki, and J. Vacata. New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda. *Acta Technica*, 56:447–466, 2011.

[12] T. Oberhuber, A. Suzuki, and V. Žabka. The cuda implentation of the method of lines for the curvature dependent flows. *Kybernetika*, 47:251–272, 2011.

[13] T. Oberhuber, A. Suzuki, J. Vacata J. V. Žabka, Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods, *Journal of Math-for-Industry*, 2011, vol. 3, pp. 73–79

[14] M. J. Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw Hill, 2003.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[16] V. Volkov and J. Demel. LU, QR and Cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-49, Electrical Engineering and Computer Sciences University of California at Berkeley, 2008.

[17] G. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, 1990.