Acta Technica *56* (2011), 447–466 © 2011 Institute of Thermomechanics AS CR, v.v.i.

New Row-grouped CSR format for storing sparse matrices on GPU with implementation in CUDA[1]

Tomáš Oberhuber[2,3], Atsushi Suzuki[4], Jan Vacata[2]

Abstract. A new format for storing sparse matrices is suggested. It is designed to perform well mainly on GPU devices. Its implementation in CUDA is presented. Its performance is tested on 1600 different types of matrices. This format is compared in detail with a hybrid format, and strong and weak points of both formats are shown.

Key words. Sparse matrices, SpMV, parallel computing, GPU, thread computing, CUDA.

1. Introduction

Graphics processing units (GPUs) are understood nowadays rather as high performance computational devices than only computer graphics accelerators. Their peak performance is beyond 1 TFLOPS in case of the single precision arithmetic. Comparison of wide class of problems solved on the CPU and GPU can be found in [1]. In this text, we concentrate on a kernel code of the numerical linear algebra. The dense matrix operations are successfully covered by BLAS implemented on GPU, e.g. CUBLAS [2]. Dense linear system solvers are presented in [3], [4]. For the sparse matrices, solvers based on the Krylov

[1]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 Supercomputing Methods in Mathematical Modelling of Problems in Engineering and Natural Sciences, project of the Student Grant Agency of the Czech Technical University in Prague No. 283 OHK4-009/10 P3913.

[2]Department of mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, 120 00 Praha 2, Czech Republic

[3]E-mail: `tomas.oberhuber@fjfi.cvut.cz`

[4]Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, Boîte courrier 187, 75252 Paris Cedex 05, France

subspaces methods are usually used. These iterative solvers spend most of the time by computing product of the matrix and vector. This operation is denoted as SpMV (Sparse Matrix Vector multiplication). Implementing iterative solvers like CG or GMRES on the GPU mainly requires having implemented SpMV (Sparse Matrix Vector multiplication), SAXPY (Scalar Alpha X Plus Y) and SDOT (Scalar DOT product) operations on GPU. The rest of the solver remains the same as for the CPU implementation. In this article we deal with the SpMV operation. SAXPY and SDOT are rather simple operations and belong to BLAS level 1.

The GPU devices can profit from their great performance only in the case of arithmetically intensive algorithms. Arithmetic intensity is the ratio of the number of arithmetic operations to memory accesses. SpMV operation $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ for the sparse matrix $\boldsymbol{A}$ requires one multiplication and one addition for each non-zero element. We need to read at least one non-zero element from the matrix $\boldsymbol{A}$ and one element from the vector $\boldsymbol{x}$. The arithmetic intensity cannot be higher than one. It means that we are not bounded by the arithmetic performance of the GPU, but by the memory bandwidth.

Formats for storing sparse matrices often involve additional information which must be read. Sometimes, data need to be aligned for faster transfer, which means adding artificial zero elements. Both additional information and artificial zeros increase the amount of data to transfer and slow down the SpMV operation. On the other hand, depending on the sparse matrix pattern, elements of the vector $\boldsymbol{x}$ may be accessed repetitively. Caching of the vector $\boldsymbol{x}$ can improve the performance significantly. An efficient format for the sparse matrices should satisfy the following:

- storing the data in continuous blocks,
- storing as less data as possible,
- reusing data of the vector $\boldsymbol{x}$.

This problem is relatively simple to be solved when $\boldsymbol{A}$ is well structured, for example, if $\boldsymbol{A}$ is multidiagonal [5]. For SpMV operation of general matrices, techniques for cache utilizations by block access [6] and data compression for both index and value [7] were introduced. On the GPUs, the work [8] studied the SpMV operation for general matrix and paper [9] for sparse matrices appearing in the graph mining. A hybrid format was proposed by [5] and extended to a block version by [10].

1.1. Contributions

We modify the common CSR (Compressed Sparse Rows) format to run efficiently on the GPU. The format that we obtain is simple as well as the kernel for the SpMV operation. We have tested this format on 1 600 sparse matrices from [11], [12]. We present several statistics following from our experiments.

We compare the new format with the Hybrid format [5] and show strong and weak points of both algorithms.

1.2. Organization

The article is organized as follows. In Sec. 2 we explain the necessary know-how of the CUDA device, which is *de facto* the standard of the GPU computing device. We establish the conditions that should be fulfilled by the algorithm to gain maximum performance in SpMV operation on CUDA device. In Sec. 3 we show already existing formats and we study their advantages and disadvantages. In this section we also present the Row-grouped CSR format. The performance and comparison with the Hybrid format is the subject of Sec. 4. Here we show a detail analysis of both formats from the performance point of view with both single and double precisions.

2. CUDA architecture

CUDA (Compute Unified Device Architecture) is an architecture designed by the Nvidia company to simplify the development of applications using GPU. CUDA is restricted only to GPUs by Nvidia. Geforce GTX280 is one of the first CUDA devices capable of computations in the double precision. It is composed of 30 multiprocessors, each having eight CUDA cores executing the single precision arithmetic and one processing unit for the double precision arithmetic. Each multiprocessor is equipped with 16 kB of very fast shared memory. It stores both data and instructions. All the multiprocessors are connected to the global memory, which is understood as an SMP architecture. The size of the global memory can be up to 1 GB, but it is only balanced with 20 % of peak performance of the double precision arithmetic. There is a read-only cache memory called a texture cache, which is bound to a part of the global memory when a code starts by the multiprocessors.

From the programmer's point of view, the most important computing entity is the thread. One writes a code called kernel which is processed by many threads. The multiprocessor can process 32 threads simultaneously. Such group of threads is referred as warp. Each thread of the warp must perform the same instruction at the same time. The essential property of the CUDA threads is that they are very lightweight. The multiprocessor is, therefore, capable to hold more than 32 threads and to switch between them efficiently. This group of threads is called block. The thread scheduler decides which threads are ready to be processed. By this mechanism, the latencies of the global memory can be efficiently hidden. There can be up to 512 threads in one block. Threads can be explicitly synchronized by the programmer. Blocks are grouped into grids and execution of blocks in the grid is distributed on multiprocessors.

Fast access to global memory is essential for computation with CUDA device. This memory is well designed for sequential access but not for random access. By nature of the hardware, the global memory is accessed by every aligned 128 bytes segment and is fed to threads in a half of the warp which is called half-warp (see [13]). The way of accessing the threads in the half-warp to aligned 128 bytes segment is called a coalesced memory access. For example, sequential access to array with 32 double precision variables can be coalesced with halves. When threads in a half-warp access to scattered address in $128 \times n$ bytes, 128 bytes segments are accessed n times, and, as a result, the memory access becomes n times slower. This memory access is called non-coalesced. Access to non-aligned 128 bytes segment is not coalesced either, because two times access with 128 bytes are necessary. Therefore, the programmer must design his code to fulfill the conditions of coalesced memory access to obtain good performance. With the coalesced memory accesses we can transfer more than 140 GB/s between the global memory and the multiprocessors. If one can find data reuse in the algorithm, the shared memory is better to be utilized with explicit code to copy data from the global memory into shared memory and to write back data. Texture cache can improve non-coalesced memory access, although it is only valid for data reading.

The new matrix format that we present in this article is based on reorganization of the CSR format such that the most of the data accesses can be coalesced.

3. Sparse matrix formats for GPUs

3.1. Common CSR format

For common sequential systems the CSR format is very popular. It is because the matrix rows are easily accessible, which allows writing an efficient code in memory usage for the SpMV. The essence of the format is depicted in Fig. 1.

We store only the non-zero elements in a row-wise order. Two arrays, `values` and `columns`, whose sizes are equal to the number of nonzero elements, store

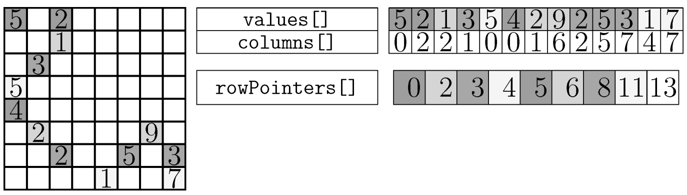

Fig. 1. CSR format

the value and column index of each element in increasing order of the column index in each row. An array `rowPointers` keeps index of array where `values` and `columns` start to keep data in the row. The size of `rowPointers` equals the size of the matrix plus one, where the last value equals the number of total nonzeros.

The code for multiplication of the sparse matrix by a vector looks like this:

```
void spmvCSR( const int rowSize,
              const REAL* values,
              const int* columns,
              const int* rowPointers,
              const REAL* x,
              REAL* Ax )
{
   for( int i = 0; i < rowSize; i ++ )
   {
      Ax[ i ] = 0.0;
      int j;
      for( j = rowPointers[ i ]; j < rowPointers[ i + 1 ]; j ++ )
         Ax[ i ] += values[ j ] * x[ columns[ j ] ];
   }
}
```

There are two possibilities to parallelize this code. Both of them are mentioned in [5]. One of them is using several threads per one row. Each thread multiplies one non-zero element of the matrix $\boldsymbol{A}$ with appropriate element of the vector $\boldsymbol{x}$. There are two main disadvantages of this approach. We must implement relatively complicated parallel reduction. Moreover, if there are only few non-zero elements on each row we do not have enough work for each thread of one warp. The results presented in [5] show that this approach does not perform reasonably. The authors refer it as vector CSR. The other way is to map one thread per each row. For large matrices there will be enough threads for efficient run on the GPU. An advantage is that we do not need to change the code for the SpMV kernel. On the other hand the way the threads read data from the arrays `values` and `columns` stored in the global memory does not fulfill the condition for the coalesced memory access. It slows down this algorithms significantly. In [5] it is referred to as scalar CSR. Both the scalar and the vector CSR are the slowest algorithms.

3.2. Blocked CSR format

The authors of [14] present Block CSR format developed for the vector-GPU architecture ATI and CTM (Close to Metal) [15]. The authors decompose the matrix into 4×4 blocks as Fig. 2 demonstrates. The disadvantage of

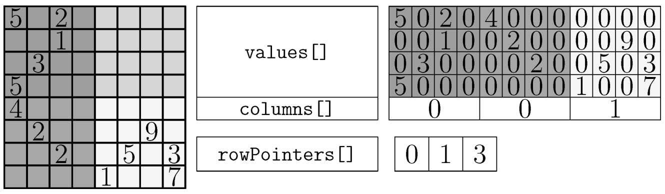

Fig. 2. Blocked CSR format

this format is that it decomposes into blocks the matrix itself and not the compressed sparse rows. Because of this, many artificial zeros appear there. As one can see in Fig. 2, there are 48 values but only 13 of them are nonzeros. The efficiency of this storage is only 27 %. All these artificial zeros must be transferred from the global memory of the GPU, which slows down the algorithm for SpMV operation. It is also wasting of the GPU global memory. Moreover, the efficiency of this format decreases with larger block size. For efficient use of the CUDA multiprocessors we would need that the block size equals 32 for coalesced memory access.

3.3. Hybrid ELLPACK and COO format

A better format for the sparse matrices storage in the GPU is the Hybrid format introduced in [5]. It is based on the combination of the ELLPACK and COO (coordinate) format. For a matrix with N rows and at most K non-zero elements per row, ELLPACK allocates NK elements in the arrays `values` and `columns`. Data storage is depicted in Fig. 3. This format works well for matrices with approximately the same number of non-zero elements in each row. Note that we do not need to store the `rowPointers`. Difficulties may appear when the number of non-zero elements differ significantly for each row. Let us consider a diagonal matrix that has one row full of non-zero elements. In this case we have $2N - 1$ non-zero elements but the ELLPACK format will store N^2 elements. Therefore the authors of [5] propose to combine the ELLPACK format with COO format. The COO format is completely explicit format storing for each non-zero matrix element its row and its column—see Fig. 4. The Hybrid format allows allocatig less than NK elements for the arrays `values` and `columns` of the ELLPACK format, and the elements that do not fit into the allocated arrays are stored in the COO format. The SpMV operation then consists of two steps, ELLPACK operation part and COO operation part. Let us say that we allocate NK_1 elements for the ELLPACK format where $K_1 < K$. If K_1 is only slightly smaller than K then we may still have a lot of artificial zeros in the ELLPACK format. If $K_1 \ll K$, we may have a lot of non-zero elements stored in the COO format. Since this format stores even the row coordinate for each non-zero element, it requires more memory than CSR format. A good choice of K_1 is essential for the Hybrid format.

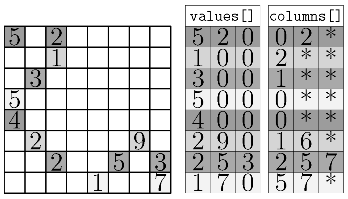

Fig. 3. ELLPACK format

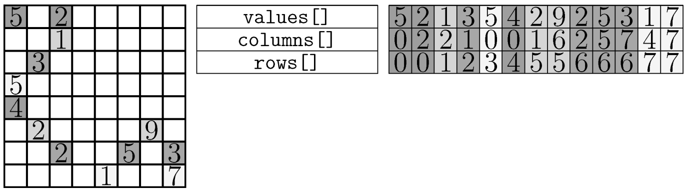

Fig. 4. COO format

3.4. Row-grouped CSR format

We present here the new Row-grouped CSR (RgCSR) format. It is a simple modification [16] of the common CSR format. Independently, a very close format, sliced ELLPACK was published by [17]. The problem is that the common CSR format does not fulfill the coalesced access to the array values and columns. Let us consider the case that 8 threads proceed SpMV operation of common CSR format in Fig. 1 with mapping of one thread to one row of the matrix. When each thread accesses to the first nonzero of each row, the positions in the arrays `values` and `columns` are 0, 2, 3, 4, 5, 6, 8 and 11. These elements are read at the same time. We see that they are not accessed sequentially in the memory.

The Row-grouped CSR format is based on storing these elements sequentially. We divide the matrix into groups of rows—see Fig. 5. In this simple example we have two groups each having four rows. In each group we store firstly the first non-zero elements in each row, then the second non-zero elements in each row, and so on. If the number of the non-zero elements differs in some row of the group, we add artificial zeros to have the same number of the elements to store in all rows of the group. In the same way, `values` array keeps original nonzeros and artificial zeros, `columns` array keeps index of column of each row and ghost index. Instead of the `rowPointers` array we store `groupPointers` and `rowLength` arrays. The former keeps the offset of the group beginning in the `values/columns` arrays. The latter keeps the number of the non-zero elements in each row. In implementation of SpMV with RgCSR format in Fig. 5, we can use 4 threads in one group.

From Figs. 2, 3 and 5 we can see that the Blocked CSR, ELLPACK and Row-grouped CSR formats allocate 35, 11 and only 7 artificial zeros, respectively.

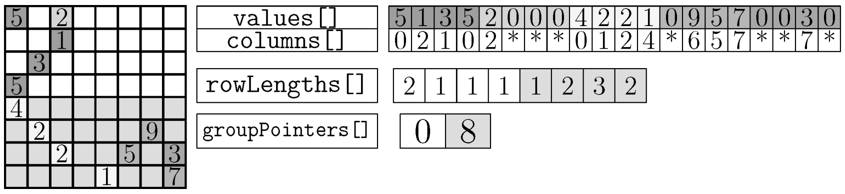

Fig. 5. Row-grouped CSR format

An advantage of the Row-grouped CSR format over the ELLPACK format is that the number of allocated elements per one row may vary from one group to another.

There is a difference between the Row-grouped CSR format and the sliced ELLPACK format. Sliced ELLPACK does not store the number of nonzeros in each row, `rowLengths[]`. The maximum number K_j of nonzeros in the j-th strip is calculated from indices of the first element of two strips, `groupPointers[j+1]` $-$ `groupPointers[j]`. It gives the same number of arithmetic operations in column direction for each strip. Sliced ELLPACK computes multiplications of zero element and pseudo-vector value to align the number of arithmetic operations per row in the strip. The RgCSR format can skip such meaningless arithmetic by using explicit information of `rowLengths[]`.

The CUDA kernel for the Row-grouped CSR format reads as follows:

```
__global__ void spmvRgCSR( const int matrixSize,
                           const REAL* values,
                           const int* columns,
                           const int* groupPointers,
                           const int* rowLengths,
                           const REAL* x,
                           REAL* Ax )
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if( row >= matrixSize ) return;

  int groupOffset = groupPointers[ blockIdx.x ];
  int ptr = groupOffset + threadIdx.x;

  // The last group may be smaller.
  int currentGroupSize = blockDim.x;
  if( ( blockIdx.x + 1 ) * blockDim. x > matrixSize )
          currentGroupSize = matrixSize % blockDim.x;

  REAL product = 0.0;
  const int rowLength = rowLengths[ row ];
  for( int i = 0; i < rowLength; i ++ )
  {
          product += values[ ptr ] * x[ columns[ ptr ] ];
          ptr += currentGroupSize;
  }
  Ax[ row ] = product;
}
```

Here, `blockDim.x` takes the size of block, which is set as the size of group. The integer variables `blockIdx.x` and `threadIdx.x` are the index of block and index of thread, respectively, the former taking the value from 0 to $\lceil$ `rowSize/ blockDim.x` $\rceil - 1$ and the latter from 0 to `blockDim.x` -1. The symbol $\lceil x \rceil$ shows the smallest integer greater than or equal to x.

It is clear that the smaller group, the less artificial zeros there are. The smallest group size which can fulfill the condition of coalesced memory access on the CUDA devices is 32, i.e. the warp size. In practical computing we usually choose larger group size.

We now show an estimation of the peak performance of the CUDA kernel for the RgCSR format. We assume the maximum memory performance to be m GB/s. For each non-zero element we need to perform one multiplication and one addition. It means that the number of floating point operations per one SpMV operation is twice as high as the number of the non-zero elements in the matrix. To process arithmetic for one non-zero element, we must read one integer from the `columns` array and two single or double precision floating point numbers, one of them coming from the array `values` and the other from the vector $\boldsymbol{x}$. For simplicity of estimating the upper bound of the performance, we omit the other arrays. Since 32 bit integer is used in CUDA GPU, data access on each step takes 12 bytes in the single precision arithmetic and 20 bytes in the double precision arithmetic. We will attain the maximum performance as $m/12$ GFLOPS for single precision and $m/20$ GFLOPS for double precision, respectively. We note that access to the array `values` is coalesced by the design of the format. However, access to the elements of the vector $\boldsymbol{x}$ is not coalesced in general. It deteriorates the real performance. For remedy of this problem, we can utilize cache memory for reading the vector $\boldsymbol{x}$. This is done by binding the vector $\boldsymbol{x}$ to a texture in CUDA device. In the ideal case of perfect data-reuse of the vector $\boldsymbol{x}$, almost all data accesses of $\boldsymbol{x}$ are cached, and we can omit reading the vector $\boldsymbol{x}$ from our estimation. This leads to 8 bytes in the single precision arithmetic and 12 bytes in the double precision arithmetic. The maximum performance will be $m/8$ GFLOPS for single precision and $m/12$ GFLOPS for double precision, respectively. Table 1 shows the estimation of the peak performance of the GTX280 card with 141 GB/s bandwidth.

Table 1. Estimation of peak performances of Row-grouped CSR format on GTX280 with 141 GB/s memory bandwidth

Texture cache for vector $\boldsymbol{x}$	Single	Double
without	23.5	14.1
with	35.25	23.5

4. Experimental evaluation

4.1. Setting of experiments

The experiments were performed on a PC equipped with Intel Core2 Quad CPU Q6700 running at 2.66 GHz with 4 MB L2 Cache, 8 GB DDR3-1333 SDRAM, and Nvidia GTX 280 card. While Nvidia GTX 280 has the peak memory bandwidth 141 GB/s, the DDR3-1333 module has 10.667 GB/s. We used CUDA toolkit ver. 3.1 to implement our Row-grouped CSR format and an implementation of the Hybrid format from CUSP library [18].

Doing the same estimation of the peak performance of the CSR format on the CPU as we did for the Row-grouped CSR format on GTX280, we get 0.89 GFLOPS for single precision and 0.53 GFLOPS for double precision without cache memory. Since the cache memory effect of the CPU is much more complicated than on the GPU, it is difficult to estimate the peak performance of the CSR format on CPU with cache memory ([19] provides an estimation for the CSR format with block access). However, we can see that GPU is much more advantageous than CPU.

We tested the common CSR, the Hybrid and the Row-grouped CSR formats on a set of 1 596 square matrices collected from two matrix markets [11], [12]. The statistics were computed in three ways—on the complete set of matrices, on small matrices with size smaller than 10 000 and on large matrices with size larger than or equal to 10 000. Table 2 shows minimum, maximum and average size, non-zero elements and ratio of nonzeros to number of the whole elements of the matrix in each set.

4.2. Performance of Hybrid format

We first show performance of the common CSR format and the Hybrid format on GTX 280 card in Table 3. We can see that the performance of the common CSR format does not depend much on the size of the matrix nor on the precision of the arithmetic. The maximum performance of the Hybrid format is 16 GFLOPS in the single precision and 11 GFLOPS in the double precision, i.e. 68 % and 78 % of the estimated performance of the GTX 280 card, respectively. We can also see that for the small matrices the average speed-up is 0.97 and 0.69, respectively. Hence, in general, it does not make sense to use

Table 2. Properties of matrix sets

	Complete set	Small matrices	Large matrices
Number of matrices	1598	1061	537
Min. size	5	5	10 000
Max. size	2 063 494	9 941	2 063 494
Average size	41 258	3 253	116 059
Min. non-zero els.	15	15	6 639
Max. non-zero els.	52 672 325	3 279 690	52 672 325
Average non-zero els.	947 367	64 899	2 684 263
Min. non-zero ratio	3×10^{-6} %	0.01 %	3×10^{-6} %
Max. non-zero ratio	100 %	100 %	2.10 %
Average non-zero ratio	1.20 %	1.76 %	0.09 %

Table 3. Performance of the common CSR format and Hybrid format in GFLOPS; speed-up of Hybrid format is measured against the common CSR format

	Complete set		Small matrices		Large matrices	
	Single	Double	Single	Double	Single	Double
CSR min.	0.16	0.1	0.1	0.1	0.19	0.19
CSR max.	1.4	1.3	1.4	1.3	1.2	1.2
CSR average	0.88	0.83	0.84	0.83	0.82	0.82
Hybrid min.	0.0009	0.0008	0.0009	0.0008	0.24	0.19
Hybrid max.	16.0	11.0	8.9	6.1	16.0	11.0
Hybrid average	2.56	1.57	1.07	0.72	5.48	3.2
Speed-up min.	0.00021	0.003	0.00021	0.003	0.63	0.48
Speed-up max.	36	10	7.2	4.9	36	11
Speed-up average	2.54	1.76	0.97	0.69	5.59	3.87

the Hybrid format for the small matrices even though it can be six times or seven times faster in some special cases. The situation is much better with the large matrices where the average speed-up is 5.59 and minimum speed-up 0.63.

4.3. Performance of RgCSR format

Table 4 shows performance and the speed-up of the Row-grouped CSR on GTX280 with respect to various group sizes from 32 to 256. It also shows the filling of the RgCSR format with artificial zeros. Let us start by commenting the filling. The 100% filling means that the amounts of the artificial zeros and the non-zero elements are the same. The best filling is 0 %. It is attained when matrix has the same number of non-zero elements in every group of rows whose size equals to the group size. We do not show this in the table. The best average value is 105 %. It means that in average we must store twice as much data as in the common CSR format. In the worst case, we store 85 times more data. We consider it the major weakness of the RgCSR format. We can see that the rate of filling almost does not depend on the matrix size, because filling is done as local operation with group size, which is much smaller than the matrix size.

Now let us to see the effect of the group size on the performance. We can see that up to the group size 128 the performance grows and it drops a little for the group size 256. Memory access with group size 32 satisfies coalesced access. However, there is another factor for faster memory access, and more than 32 threads (one warp) are necessary to hide memory latencies well. It can be measured by the warp occupancy how well the threads access the global memory (see [13]). With occupancy 1.0 multiprocessors run warps without delay caused by memory access. In this experiment, occupancy is 0.25 with 32 threads, 0.5 with 64 threads, and 1.0 with 128 and 256 threads. Due to increasing of number of filling, ratio of effective memory in the coalesced is decreasing, so we have an optimal size on the group.

In average, the difference in the performance is not significant but for the maximum performance it is, especially for the single precision. Here it grows from 20.6 GFLOPS to 32.8 GFLOPS. On the comparison of RgCSR on GPU to common CSR on CPU, RgCSR on GPU is in average 4.3 times and 3.4 times faster in single and double precisions, respectively, but 100 times slower in the worst case. When we restrict ourselves only to small matrices, these numbers decrease approximately to 2.15 and 1.9, respectively. It means that the RgCSR can be reasonably used even for small matrices for which the Hybrid format is not profitable choice. For the large matrices the RgCSR also offers a good performance, e.g., the speed-up is 8.64 and 7.94 in single and double precisions, respectively, while for the Hybrid format these speed-ups are 5.5 and 3.2, respectively.

It is worth noting that RgCSR format on CPU is capable of better performing than the common CSR. In average it gives only 55 % performance of the CSR format but in some cases RgCSR format can better use the cache and it can be up to 4 times faster than CSR. However, we will not study the RgCSR on the CPU more in this text.

Let us also comment the effect of caching the vector $\boldsymbol{x}$ by binding it to the texture memory. Without caching the best performance in the single precision was only 18.03 GFLOPS. Turning the caching on increased this number 1.81 times to 32.84. In average, the difference was up to 30 %. With the double precision the best performance grows 1.3 times from 14.46 GFLOPS to 18.82 GFLOPS, and in average the difference is 37 %.

Another important information is how far from the peak performance we are. If we take the best case, i.e., 32.8 and 18.82 GFLOPS for the single and double precisions, respectively, it makes 93 % and 80 % of the peak performance, respectively. If we omit the caching of the vector $\boldsymbol{x}$, it is 53 % and 71 %, respectively.

Table 4. Performance of RgCSR format; speed-up of RgCSR format is measured against the common CSR format running on CPU

	Complete set		Small matrices		Large matrices	
	Single	Double	Single	Double	Single	Double
	Group size 32					
Artif. zeros max.	1032 %		1032 %		870 %	
Artif. zeros average	105 %		107 %		102 %	
RgCSR min.	0.01	0.01	0.01	0.01	0.02	0.02
RgCSR max.	20.6	16.33	11.2	9.23	20.6	16.33
RgCSR average	3.19	2.72	1.7	1.5	6.09	5.14
Speed-up min.	0.01	0.01	0.01	0.01	0.02	0.02
Speed-up max.	18.75	26.2	9.1	9.1	18.75	26.2
Speed-up average	3.18	2.72	1.7	1.5	6.15	6.27
	Group size 64					
Artif. zeros max.	2098 %		2098 %		1430 %	
Artif. zeros average	144 %		137 %		157 %	
RgCSR min.	0.01	0.01	0.01	0.01	0.02	0.02
RgCSR max.	28	18.64	19	14.9	28.05	18.46
RgCSR average	4.18	3.35	2.2	1.9	8.01	6.19
Speed-up min.	0.01	0.01	0.01	0.01	0.02	0.02
Speed-up max.	25.89	27.13	15.4	14.6	25.9	27.13
Speed-up average	4.14	3.81	2.13	1.9	8.07	7.55
	Group size 128					
Artif. zeros max.	4230 %		4230 %		2476 %	
Artif. zeros average	206 %		183 %		250 %	
RgCSR min.	0.01	0.01	0.01	0.01	0.02	0.02
RgCSR max.	32.8	18.82	19.9	14.15	32.84	18.82
RgCSR average	4.38	3.43	2.2	1.87	8.58	6.52
Speed-up min.	0.01	0.01	0.01	0.01	0.02	0.02
Speed-up max.	26.98	26.22	16.1	13.94	26.98	26.22
Speed-up average	4.34	3.91	2.14	1.87	8.64	7.94
	Group size 256					
Artif. zeros max.	8494 %		8494 %		4684 %	
Artif. zeros average	304 %		255 %		400 %	
RgCSR min.	0.01	0.01	0.01	0.01	0.02	0.02
RgCSR max.	31.6	18.3	21.3	13.9	31.57	18.3
RgCSR average	4.37	3.37	2.24	1.83	8.55	6.38
Speed-up min.	0.01	0.01	0.01	0.01	0.02	0.02
Speed-up max.	26.1	25.46	17.2	13.7	26.13	25.46
Speed-up average	4.33	3.37	2.15	1.83	8.58	7.78

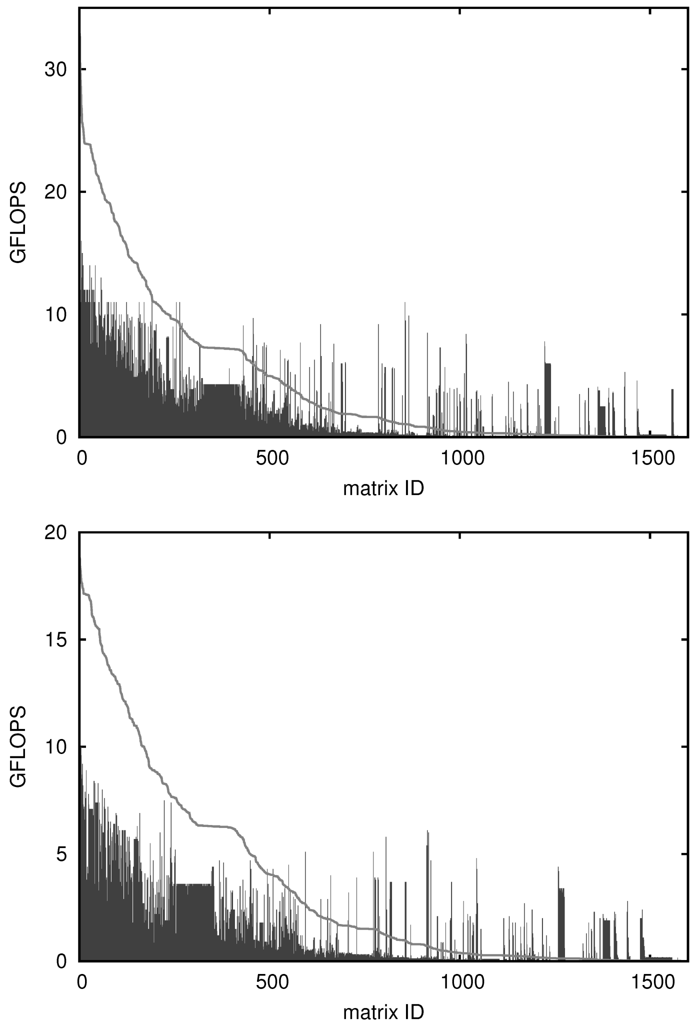

Fig. 6. Performance of RgCSR format (line) and Hybrid format (bars) in single precision (*top*) and double precision (*bottom*) on the complete set of 1 596 matrices that are sorted in descending order by the performance of RgCSR format

4.4. Comparison of RgCSR format and Hybrid format

We show the comparison of RgCSR format to other formats. Here we fixed group size as 128, which attains the best results in Table 4.

4.4.1. Outlines from the set of matrices. Table 5 shows the comparison of the best RgCSR setting and the Hybrid format. It shows in how many cases the Hybrid format is faster than the CSR format and the same for the RgCSR vs.

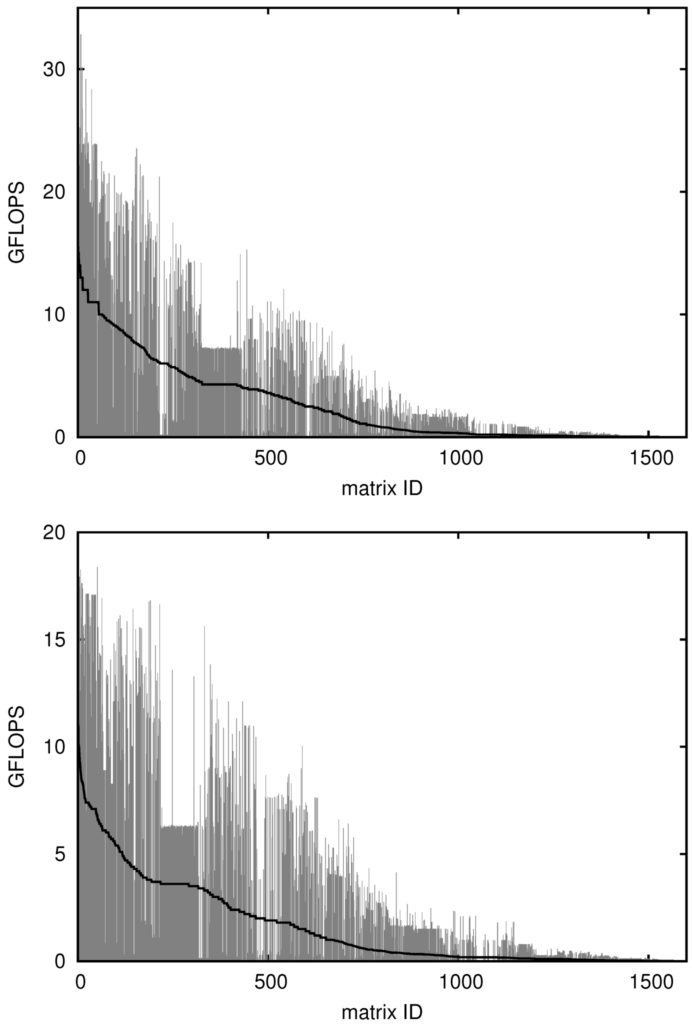

Fig. 7. "Inverse" figure to Fig. 6: performance of RgCSR format (bars) and Hybrid format (line) in single precision (*top*) and double precision (*bottom*) on the complete set of 1 596 matrices that are sorted in descending order by the performance of Hybrid format

CSR format and the RgCSR vs. the Hybrid format. It also shows the average speed-up of the RgCSR format related to the Hybrid format. Here we can see again that the RgCSR format outperforms the Hybrid format well with the small matrices. With the large matrices the RgCSR format is 1.8 times and 2.18 times faster in double and single precisions, respectively, than the Hybrid format.

Figures 6 and 7 show detailed performance comparison of both RgCSR and the Hybrid formats with all 1 596 matrices by graphs whose horizontal axis

Table 5. Comparison of CSR, RgCSR and Hybrid formats; the first three rows below the head say for how many matrices is a format faster than the others, the fourth row shows average speed-up of RgCSR format against Hybrid format

	Complete set		Small matrices		Large matrices	
	Single	Double	Single	Double	Single	Double
HYB faster than CSR	48.17 %	42.63 %	22.32 %	16.71 %	98.70 %	93.64 %
RgCSR faster than CSR	56.68 %	55.79 %	46.34 %	45.20 %	76.70 %	76.64 %
RgCSR faster than HYB	77.14 %	80.67 %	84.43 %	85.40 %	62.57 %	71.40 %
Average RgCSR/HYB	2.55	3.21	3.21	3.74	1.24	2.18

is based on sorted matrix-ID according to either performance. In the single precision arithmetics there are only 3 matrices for which RgCSR achieves more than 30 GFLOPS, 30 matrices for which RgCSR gets over 20 GFLOPS and 200 matrices with the performance over 10 GFLOPS. There are only 60 matrices for which the Hybrid format gets over 10 GFLOPS. Similar results can be observed even with the double precision—there are 170 matrices for which the RgCSR performs better than 10 GFLOPS and only 4 matrices for which the Hybrid format gets over 10 GFLOPS. We observe the following tendency in double precision from the graph of Fig. 6-*bottom*. Matrices whose processing speed with RgCSR is grater than 4 GFLOPS allow faster computation than the Hybrid format. On the other hand, in case of matrices which obtain only less than 1 GFLOPS by RgCSR, the Hybrid format has possibility to perform faster than RgCSR.

4.4.2. Detailed comparison with specific matrices. We show now four matrices from [11], which produce significant differences in performance using the RgCSR and the Hybrid formats. The matrix names are `Hohn/fd18`, `AMD/G2_circuit`, `IBM_EDA/trans4`, and `Rajat/Raj1`, whose nonzero patterns are shown in Fig. 8. Characters of matrices and performances in double precision are summarized in Table 6. RgCSR runs with group size 128 and texture caching on. While `Hohn/fd18` and `AMD/G2_circuit` have a small number of nonzeros in each row and RgCSR performs very well, `IBM_EDA/trans4` and `Rajat/Raj1` have large variations in number of nonzeros and RgCSR performs very badly. We call `Hohn/fd18` and `AMD/G2_circuit` the first group of the four matrices and the rest the second group. For more detailed analysis of performance of RgCSR, we employed ordering of row index of the matrix. Employing an appropriate ordering of row index, RgCSR can reduce the number of artificial nonzeros for alignment of array in each group. We used the simplest ordering, descending order of number of nonzeros in row, and AMD ordering (approximate minimum degree ordering) [20], which can reduce fill-in during LU factorization. Descending ordering is the optimal way of suppressing artificial nonzeros but it may shuffle nonzeros pattern of the matrix. On the other hand, AMD ordering can reduce the range of off-diagonal distribution of

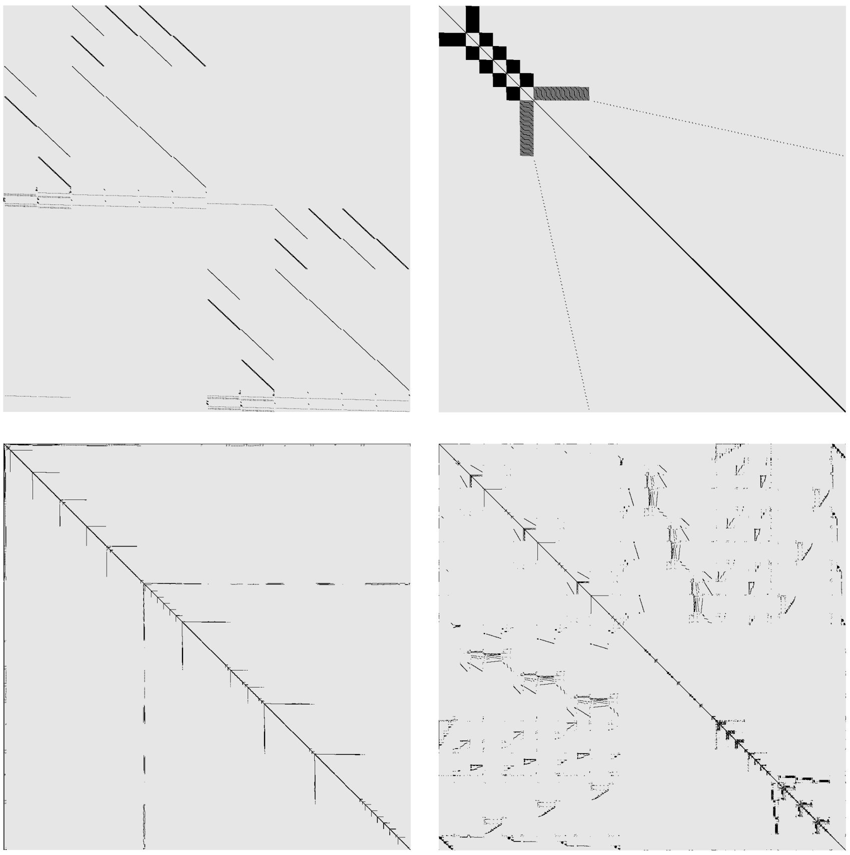

Fig. 8. Non-zero patterns of matrices; *top-left*: Hohn/fd18, *top-right*: AMD/G2_circuit, *bottom-left*: IBM_EDA/trans4, *bottom-right*: Rajat/Raj1

the matrix elements. The result is summarized in Table 7. The third row of each sub-table shows use of the texture cache with hit and missed cases, which was measured by CUDA profiler tool. The sum of cache hit and missed cases is proportional to the number of nonzeros of the matrix. We can see that the ratios of cache miss are even higher in the first group than in the second group. Decreasing ordering can reduce artificial nonzeros drastically, which helps to reduce the memory requirement. AMD ordering shows a better use of texture cache than the others but it suffers from larger artificial nonzeros than the descending ordering. By this comparison of cache miss rate on two groups of matrices, we can see that the ratio of cache hit cases is not the leading term on the performance.

By introducing artificial nonzeros to align size of data in columns per each group of rows, memory access inside of each multiprocessor is coalesced, but memory access among multiprocessors is still unaligned in the second group

Table 6. Characters of matrices and performances in double precision with the RgCSR and Hybrid formats, with performance of the CSR format on CPU.

Matrix name	Number of rows	Number of nonzeros in row			GFLOPS		Ratio	CPU CSR
		max	mean	min	RgCSR	Hybrid		
`Hohn/fd18`	16 248	6	3.860	1	4.69	0.95	4.93	1.05
`AMD/G2_circuit`	150 102	6	4.841	2	9.36	2.5	3.74	0.60
`IBM_EDA/trans4`	116 835	114 190	6.600	1	0.019	2.0	0.095	0.59
`Rajat/Raj1`	263 743	40 468	4.938	1	0.058	2.2	0.026	0.50

Table 7. Effect of ordering to number of artificial nonzeros and performance of RgCSR.

	Ordering		
	none	descending	AMD
`Hohn/fd18`	16 248 rows, 63 406 nonzeros		
Artif. zeros	2.76 %	0.34 %	26.38 %
GFLOPS	4.690	4.845	3.900
cache hit/miss	289/1 350	324/1 325	312/2 907
`AMD/G2_circuit`	50 102 rows, 726 674 nonzeros		
Artif. zeros	3.90 %	0.05 %	2.03 %
GFLOPS	9.364	9.210	8.044
cache hit/miss	431/12 026	622/11 592	3 448/11 951
`IBM_EDA/trans4`	116 835 rows, 766 396 nonzeros		
Artif. zeros	2 118.1 %	1 452.8 %	1 613.3 %
GFLOPS	0.0189	0.0191	0.0188
cache hit/miss	9 213/16 347	6 517/12 576	11 295/14 727
`Rajat/Raj1`	263 743 rows, 1 302 464 nonzeros		
Artif. zeros	938.2 %	189.3 %	370.6 %
GFLOPS	0.0578	0.0904	0.0808
cache hit/miss	9 229/17 049	6 653/12 513	10 793/11 707

matrices due to large variation in the number of nonzeros. This is the reason why the second group matrices suffer very poor performance even much worse than the common CSR format on CPU. Use of large size of group to achieve aligned access by 30 multiprocessors, such as 128×30, leads unfortunately to a very large number of artificial nonzeros. This is a true weak point of the RgCSR format. For matrices enjoying a good performance by the RgCSR format, there is a possibility of further improvement of performance by decreasing the number of artificial nonzeros and increasing cache utilization by means of the ordering of row index.

5. Conclusion

We proposed Row-grouped CSR format to store a sparse matrix, which can run efficiently on the GPU with continuous data access called coalesced access in the terminology of the CUDA GPU architecture. We verified that RgCSR format can perform better than the Hybrid format by numerical experiments using 1 600 matrices. However, on some matrices, RgCSR format performs very poorly due to a complicated pattern of nonzero elements of matrix, even though the Hybrid format can perform close to its average speed. For enhancement of performance of RgCSR format, a good ordering of index of the matrix row is necessary, by which the usage of texture cache fetching the right-hand vector is improved. This will be the subject of our future research.

The source code of the RgCSR format is available as a part of the Template Numerical Library (TNL) at `http://geraldine.fjfi.cvut.cz/~oberhuber/doku-wiki-tnl`.

References

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, P. Dubey: *Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU.* Proc. 37th Ann. Int. Symposium on Computer Architecture (ISCA'10) Saint-Malo (France), June 19–23, 2010, ACM, New York 2010, 451–460.

[2] NVIDIA Corporation, CUDA CUBLAS library, PG-00000-002_V3.1, May 2010, `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/CUBLAS_Library_3.1.pdf`.

[3] N. Galoppo, N. K Govindaraju, M. Henson, D. Manocha: *LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware.* Proceedings ACM/IEEE SC'05, Conference of Supercomputing, Nov. 12–18, 2005, Seattle (USA), doi: 10.1109/SC.2005.42.

[4] V. Volkov, J. Demel: *LU, QL and Cholesky factorizations using vector capabilities of GPUs.* Techn. Rep. UCB/EECS-2008-49, Electrical Engineering and Computer Sciences, University of California, Berkeley, 2008.

[5] N. Bell, M. Garland: *Efficient sparse matrix-vector multiplication on CUDA.* Techn. Rep. NVR-2008-004, NVIDIA Corporation 2008.

[6] E.-J. Im: *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, Rep. UCB/CSD-00-1104, University of California, Berkeley, 2000.

[7] K. Kourtis, G. Goumas, N. Koziris: *Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression.* Proc. 37th International Conference on Parallel Processing, Sept. 8–12, 2008, Portland (USA), 511–519.

[8] M. M. Baskaran, R. Bordwaker: *Optimizing sparse matrix-vector multiplication on GPUs.* Techn. Rep. RC24704(W0812-047), IBM 2008, `http://domino.watson.ibm.com/library/CyberDig.nsf/papers/1D32F6D23B99F7898525752200618339/$File/rc24704.pdf`.

[9] J. W. Choi, A. Singh, R. Vuduc: *Model-driven autotuning of sparse matrix-vector multipy on GPUs.* Proc. 15th ACM SIGPLAN Symposium on Principles and Practice

of Parallel Programming (PPoPP 2010), Bangalore (India), Jan. 9–14, 2010 (R. Govindarajan, D. A. Padua, M. W. Hall, eds.), ACM 2010, 37–48.

[10] A. Monakov, A. Avetisyan: *Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs.* Proc. 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos (Greece), July 20–23, 2009 (K. Bertels, N. J. Dimopoulos, C. Silvano, S. Wong, eds.) Springer, Berlin 2009, 289-297.

[11] T. A. Davis, Y. Hu: *The University of Florida sparse matrix collection.* NA Digest 92 (42), `http://www.cise.ufl.edu/research/sparse/matrices/`.

[12] Z. Bai, D. Day, J. Demmel, J. Dongarra: *Test matrix collection (non-Hermitian eigenvalue problems).* release 1, Techn. Rep., University of Kentucky, 1996, `http://math.nist.gov/MatrixMarket/`

[13] *NVIDIA CUDA Programming Guide 3.0.* NVIDIA Corporation, 2010, `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`.

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

[15] *Amd "close to metal" technology unleashes the power of stream computing.* Techn. Rep., AMD Press Release 2006.

[16] J. Vacata: *GPGPU: General purpose computation on GPUs.* Master's thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, 2008.

[17] A. Monakov, A. Lokhmotov, A. Avetisyan: *Automatically tuning sparse matrix-vector multiplication for GPU architectures.* Proc. 5th International Conferences on High Performance Embedded Architectures and Compilers (HiPEAC 2010), Pisa (Italy), Jan. 25–27, 2010 (Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, X. Martorell, eds.), Springer, Berlin 2010, 111–125.

[18] *Nvidia, Cusp 0.1.1.* `http://code.google.com/p/cusp-library/`, 2010.

[19] R. Nishtala, R. W. Vuduc, J. W. Demmel, K. A. Yelick: *When cache blocking of sparse matrix vector multiply works and why.* Appl. Algebra Eng. Commun. Comput. *18* (2007), 297–311.

[20] P. Amestoy, T. A. Davis, I. S. Duff: *An approximate minimum degree ordering algorithm.* SIAM J. Matrix Anal. Appl. *17* (1996), 886–905.

Received March 12, 2011