New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA.

Tomáš Oberhuber\textsuperscript{a}, Atsushi Suzuki\textsuperscript{b}, Jan Vacata\textsuperscript{a},

\textsuperscript{a}Department of mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, Praha 2, 120 00, Czech Republic
\textsuperscript{b}Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie, Boîte courrier 187, 75252 Paris Cedex 05, France

Abstract

In this article we present a new format for storing sparse matrices. The format is designed to perform well mainly on the GPU devices. We present its implementation in CUDA. The performance has been tested on 1,600 different types of matrices and we compare our format with the Hybrid format proposed in \cite{1}. We give detailed comparison of both formats and show their strong and weak parts.

Keywords: sparse matrices, SpMV, parallel computing, GPU, thread computing, CUDA
65F50, 65F10, 65Y05

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 the GPU can be found in \cite{2}. 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 \cite{3}. Dense linear system solvers are presented in \cite{4, 5}. For the sparse matrices, solvers based on the Krylov subspaces methods are usually used. This kind of 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 a ratio of the number of arithmetic operations to memory accesses. SpMV operation $y = Ax$ for the sparse matrix $A$ requires one multiplication and one addition for each non-zero element. We need to read at least one non-zero element from the matrix $A$ and one element from the vector $x$. The arithmetic intensity can not 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 the sparse matrices often involves additional information which must be read. Sometimes data need to be aligned for faster transfer which means adding artificial zero elements. Both 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 $x$ may be accessed repetitively. Caching of the vector $x$ can improve the performance significantly. Efficient format for the sparse matrices should satisfy the following:

- store the data in continuous blocks
- store as less data as possible
- reuse data of the vector $x$

This problem is relatively simple to solve when $A$ is well structured, for example if $A$ is multi-diagonal [1]. For SpMV operation of general matrices, techniques for cache utilizations by block access [7] and data compression for both index and value [8] were introduced. On the GPUs [9] studied the SpMV operation for general matrix and [10] for sparse matrices appearing in the graph mining. Hybrid format was proposed by [1] and was extended to a block version by [11].

1.1. Contributions

We modify common CSR format to run efficiently on the GPU. The format we obtain is simple as well as the kernel for the SpMV operation. We have tested this format on 1600 sparse matrices from [12, 13]. We present several statistics made from our experiments. We compare the new format with the Hybrid format [1] and show strong and weak parts of both algorithms.

1.2. Organization

The article is organized as follows. In the Section 2 we explain the necessary knowledge of CUDA device, which is a de facto standard of GPU computing device. We establish what conditions should be fulfilled by the algorithm to gain maximum performance in SpMV operation on CUDA device. In the Section 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 subject to the Section 4. Here we show detail analysis of both formats from the performance point of view with both single and double precision.

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 very fast 16 KB of shared memory. It stores both data and instructions. All the multiprocessors are connected to the global memory, which is understood as an SMP architecture. Size of the global memory can be up to 1GB 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 as half-warp (see [14]). The way how the the threads in the half-warp access to aligned 128 bytes segment is called as 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 the result, memory access becomes n times slower. This memory access is called as 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, 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 (Compressed Sparse Rows) format is very popular. It is because the matrix rows are easily accessible and it allows to write efficient code in memory usage for the SpMV. The essence of the format is depicted on the Figure[1]. We store only the non-zero elements in a rowwise order. Two arrays, values and columns, whose size is equal to the number of nonzero elements, store value and column index of the 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 rowPointers has the same size of the matrix plus one, where the last value equals the number of total non-zeros.

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

```c
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 [1]. The first one is to use several threads per one row. Each thread multiplies one non-zero element of the matrix $A$ with appropriate element of the vector $x$. 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 [1] show that this approach does not perform reasonably. The authors refer it as vector CSR. The second 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 \texttt{values} and \texttt{columns} stored in the global memory does not fulfill the condition for the coalesced memory access. It slows down this algorithms significantly. In [1] it is referred as scalar CSR. Both the scalar and the vector CSR are the slowest algorithms.

3.2. Blocked CSR format

The authors of [15] present Block CSR format developed for the vector-GPU architecture ATI and CTM (Close to Metal) [16]. The authors decompose the matrix into 4x4 blocks as the Figure 2 demonstrates. The disadvantage of 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 on the Figure 2 there 48 values but only 13 of them are non-zero. 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 block size equals 32 for coalesced memory access.
3.3. Hybrid ELLPACK and COO format

Better format for the sparse matrices storage in the GPU is the Hybrid format introduced in [1]. It is based on the combination of the ELLPACK and COO format. For matrix with $N$ rows and with maximum $K$ non-zero elements per row, ELLPACK allocates $NK$ elements in the arrays values and columns. Data storage is depicted on the Figure 3. This format works well for matrices with approximately the same number of non-zero elements on 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. Imagine that we have diagonal matrix which 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 [1] propose to combine the ELLPACK format with COO (coordinate) format. The COO format is completely explicit format storing for each non-zero matrix element its row and its column - see Figure 4. The Hybrid format allows to allocate less than $NK$ elements for the arrays values and columns of the ELLPACK format and those elements which 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’s 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 $K_1 \ll K$ then 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. Good choice of $K_1$ is essential for the Hybrid format.

3.4. Row-grouped CSR format

We present here new Row-grouped CSR format (we use RgCSR as its abbreviation). It is a simple modification [17] of the common CSR format. Independently, a very close format, sliced ELLPACK was published by [18]. The problem is that the common CSR format do not fulfill the coalesced access to the array values and columns. Let’s consider the case that 8 threads proceed SpMV
operation of common CSR format in Figure 1 with mapping of one thread to one row of the matrix. When each thread accesses to the first non-zero 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 the Figure 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. As the same way values array keeps original non-zeros 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 first one keeps the offset of the group beginning in the values/columns arrays. The second one keeps the number of the non-zero elements in each row. In implementation of SpMV with RgCSR format in Figure 5, we can use 4 threads to one group.

From the Figures 2, 3 and 5 we see that the Blocked CSR format allocates 35 artificial zeros, the ELLPACK format allocates 11 and the Row-grouped CSR only 7 of them. 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 non-zeros
in each row, \texttt{rowLengths[]} . Maximum number of non-zeros in \( j \)-th strip, \( K_j \) is calculated from indices of the first element of two strips, \texttt{groupPointers}[j + 1] \(-\texttt{groupPointers}[j] \). It gives uniform number of arithmetics in column direction for each strip. Sliced ELLPACK computes multiplications of zero-element and pseudo vector value to align arithmetic amount per row in the strip. \texttt{RgCSR} format can skip such meaningless arithmetic by using explicit information of \texttt{rowLengths[]} .

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

```c
__global__ void spmvRgCSR(const int rowSize,
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 >= rowSize ) 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 = rowSize % 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, \texttt{blockDim.x} takes size of block, which is set as the size of group. Integer variables \texttt{blockIdx.x} and \texttt{threadIdx.x} are index of block whose takes between 0 to \( \lceil \texttt{rowSize}/\texttt{blockDim.x} \rceil -1 \), and index of thread having 0 to \texttt{blockDim.x} -1, respectively. The mathematical symbol \( \lceil x \rceil \) shows the smallest integer grather or equal to \( x \).

It is clear that the smaller group we have 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 groups size.

We now show estimation of the peak performance of the CUDA kernel for the
RgCSR format. We assume the maximum memory performance as $m$ GB/sec. 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 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, where one comes from the array values and one from the vector $x$. For simplicity in estimation of 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, but the access to the vector $x$ data is not coalesced in general. Therefore, real performance is deteriorated by non-coalesced access to the $x$ data. For remedy of this problem, we can utilize cache memory for reading vector $x$. This is done by binding the vector $x$ to a texture in CUDA device. In the ideal case of perfect data-reuse of the vector $x$, almost all data accesses of $x$ are cached and we can omit reading the vector $x$ 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 estimation of the peak performance of the GTX280 card with 141 GB/s bandwidth.

4. Experimental evaluation

4.1. Setting of experiments

The experiments were performed on the PC equipped with Intel Core2 Quad CPU Q6700 running at 2.66GHz with 4MB 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 \[19\].

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

| texture cache for vector $x$ | single | double |
|-------------------------------|--------|--------|
| without                       | 23.5   | 14.1   |
| with                          | 35.25  | 23.5   |

Table 1: Estimation of peak performances of the Row-grouped CSR format on GTX280 with 141GB/sec memory bandwidth.
Table 2: Properties of the 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          | $3 \cdot 10^{-6}$% | 0.01%         | $3 \cdot 10^{-6}$% |
| Max. non-zero ratio          | 100%         | 100%           | 2.10%          |
| Average non-zero ratio       | 1.20%        | 1.76%          | 0.09%          |

GFLOPS for the single precision and 0.53 GFLOPS for the 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 ([6] provides an estimation for the CSR format with block access). However, we can see there is big advantage of GPU to CPU.

We have 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 [12, 13]. 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 or equal 10,000. The Table 2 shows minimum, maximum and average size, non-zero elements and ratio of non-zeros 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 see that the performance of the common CSR format does not depend much neither 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 resp. 11 GFLOPS in the double precision. It is 68% resp. 78% of the estimated performance of the GTX 280 card. We also see that for the small matrices the average speed-up is 0.97 resp. 0.69. So, in general it does not make sense to use the Hybrid format for the small matrices even though it can be 6 or 7 times faster in some special cases. The situation is much better with the large matrices where the average speed-up is 5.59 and it is 0.63 in the worst case.
|                      | 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      |

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

4.3. Performance of RgCSR format

The Table 4 shows performance and the speed-up of the Row-grouped CSR on GTX280 with respect to various group sizes 32 to 256. It also shows the filling of the RgCSR format with artificial zeros. Let us start by commenting the filling. 100% filling means that there is the same amount of the artificial zeros as the non-zero elements. 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 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 the common CSR format. In the worst case, we store 85 times more data. We see this as 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. How well the threads access the global memory can be measured by the warp occupancy (see [14]). 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 so 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 4.3 times resp. 3.4 times faster.
in average, but 100 times slower in the worst case. When we restrict ourselves only to small matrices these numbers decrease to 2.2 resp. 1.87. 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 good performance, e.g., speed-up is 8.64 resp. 7.94, while 5.5 resp. 3.2 of Hybrid format.

We would like to note that RgCSR format on CPU has possibility to perform better 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 $x$ 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 resp. 18.82 GFLOPS for the single resp. the double precision, it makes 93% resp. 80% of the peak performance. If we omit the caching of the vector $x$ it is 53% resp. 71%.

4.4. Comparison of RgCSR format and Hybrid format

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

4.4.1. Outlines from the set of matrices

The 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. 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 again see, that the RgCSR format outperforms the Hybrid format well with the small matrices. With the large ones the RgCSR format is 1.2 resp. 2.18 times faster 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 $x$-axis is based on sorted matrix-ID according to either performance. In the single precision arithmetics there are only few matrices were RgCSR achieves more than 30 GFLOPS (to be more specific there are only 3 of them), there are 30 matrices where RgCSR gets
|                  | Group size 32 | Group size 64 | Group size 128 | Group size 256 |
|------------------|---------------|---------------|---------------|---------------|
|                  | Complete set  | Small matrices| Large matrices|               |
|                  | Single        | Double        | Single        | Double        |
| Artif. zeros max.| 1032%         | 1032%         | 870%          |               |
| Artif. zeros avg.| 105%          | 107%          | 102%          |               |
| RgCSR min.       | 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 avg.       | 3.19          | 2.72          | 1.7           | 1.5           | 6.09          | 5.14          |
| Speed-up min.    | 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 avg.    | 3.18          | 2.72          | 1.7           | 1.5           | 6.15          | 6.27          |
| Artif. zeros max.| 2098%         | 2098%         | 1430%         |               |
| Artif. zeros avg.| 144%          | 137%          | 157%          |               |
| RgCSR min.       | 0.01          | 0.01          | 0.01          | 0.02          | 0.02          |
| RgCSR max.       | 28            | 18.64         | 19            | 14.9          | 28.05         | 18.46         |
| RgCSR avg.       | 4.18          | 3.35          | 2.2           | 1.9           | 8.01          | 6.19          |
| Speed-up min.    | 0.01          | 0.01          | 0.01          | 0.02          | 0.02          |
| Speed-up max.    | 25.89         | 27.13         | 15.4          | 14.6          | 25.93         | 27.13         |
| Speed-up avg.    | 4.14          | 3.81          | 2.13          | 1.9           | 8.07          | 7.55          |
| Artif. zeros max.| 4230%         | 4230%         | 2476%         |               |
| Artif. zeros avg.| 206%          | 183%          | 250%          |               |
| RgCSR min.       | 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 avg.       | 4.38          | 3.43          | 2.2           | 1.87          | 8.58          | 6.52          |
| Speed-up min.    | 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 avg.    | 4.34          | 3.91          | 2.14          | 1.87          | 8.64          | 7.94          |
| Artif. zeros max.| 8494%         | 8494%         | 4684%         |               |
| Artif. zeros avg.| 304%          | 255%          | 400%          |               |
| RgCSR min.       | 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 avg.       | 4.37          | 3.37          | 2.24          | 1.83          | 8.55          | 6.38          |
| Speed-up min.    | 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 avg.    | 4.33          | 3.37          | 2.15          | 1.83          | 8.58          | 7.78          |

Table 4: Performance of the RgCSR format. Speed-up of the RgCSR format is measured against the common CSR format running on the CPU.
Figure 6: Performance of both the RgCSR (red line) and the Hybrid (black bars) formats in the single (top) and the double (bottom) precision on the complete set of 1,596 matrices. The x-axis is the matrix ID and the y-axis is the performance in GFLOPS. The matrices are sorted in the descending order by the performance of the RgCSR format.
Figure 7: “Inverse” figure to the Figure 7. Performance of the RgCSR (black line) and the Hybrid (red bars) formats in the single (top) and the double (bottom) precision on the complete set of 1,596 matrices. The x-axis is the matrix ID and the y-axis is the performance in GFLOPS. The matrices are sorted in the descending order by the performance of the Hybrid format.
Complete set | Small matrices | Large matrices
---|---|---
HYB faster than CSR | 48.17% | 22.32% | 98.70%
RgCSR faster than CSR | 56.68% | 46.34% | 76.70%
RgCSR faster than HYB | 77.14% | 84.43% | 62.57%
Average. RgCSR/HYB | 2.55 | 3.21 | 1.24

Table 5: Comparison of the CSR, RgCSR and the Hybrid format. The first three rows say for how many matrices is one format faster than the others. It also shows average speed-up of the RgCSR format against the Hybrid format.

| Matrix name          | # of rows | # of nonzeros in row | GFLOPS | CPU |
|----------------------|-----------|----------------------|--------|-----|
|                       | max | mean | min | RgCSR | Hybrid | ratio | CSR |
| Hohn/fd18             | 16,248 | 6 | 3.860 | 4.69 | 0.95 | 4.93 | 1.05 |
| AMD/G2.circuit        | 150,102 | 6 | 4.841 | 9.36 | 2.5 | 3.74 | 0.60 |
| IBM_EDA/trans4         | 116,835 | 114,190 | 6.600 | 0.019 | 2.0 | 0.095 | 0.59 |
| Rajat/Raj1             | 263,743 | 40,468 | 4.938 | 0.058 | 2.2 | 0.026 | 0.50 |

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

over 20 GFLOPS and 200 matrices with the performance over 10 GFLOPS. There are only 60 matrices where 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 Figure 6, bottom. Matrices whose computation 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 [12], which produce significant differences in performance of the RgCSR and the Hybrid formats perform. The matrix names are Hohn/fd18, AMD/G2.circuit, IBM_EDA/trans4, and Rajat/Raj1, whose nonzero pattern is shown in Figure 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 small number of nonzeros in each row and then 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 as the first group of the four matrices and the rest as the second group. For more detailed analysis of performance of RgCSR, we employed ordering of row index of the matrix.
Figure 8: Non-zero patterns of matrices: the top-left is Hohn/fd18, the top-right is AMD/G2_circuit, the bottom-left is IBM_EDA/trans4, and the bottom-right is Rajat/Raj1.
Employing good ordering of row index, RgCSR can reduce 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) \cite{20}, which can reduce fill-in during LU factorization. Descending ordering is an optimal way to suppress artificial nonzeros but it may shuffle non-zeros pattern of the matrix. On the other hand, AMD ordering can reduce range of off-diagonal distribution of the matrix. The result is summarized in Table 7. The third row of each matrix shows use of texture cache with hit and missed cases, which was measured by CUDA profiler tool. Sum of cache hit and missed cases is proportional to number of the nonzeros of the matrix. We can see ratios of cache miss in the first group are even higher than in the second group. Decreasing ordering can reduce artificial nonzeros drastically which helps reduction of memory requirement. AMD ordering shows better use of texture cache than others, but suffers from larger artificial nonzeros than descending ordering. By this comparison of cache miss rate on two groups of matrices, we can see 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 matrices due to large variation in numbers of nonzero. This is the reason why the second group matrices suffer very poor performance even much worse than common CSR format on CPU. Use of large size of group to achieve aligned access by 30 multiprocessors, such as \(128 \times 30\), leads unfortunately to very large number of artificial nonzeros. This is a true weak point of the RgCSR format. For matrices enjoying good performance by the RgCSR format performs well, there is a possibility of futher improvement of performance by decreasing number of artificial nonzeros and increasing cache utilization by means of ordering of row index.

5. Conclusion

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

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

18
Table 7: Effect of ordering to number of artificial nonzeros and performance of RgCSR.

Acknowledgement

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.

References

[1] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on cuda, Tech. Rep. Technical Report NVR-2008-004, NVIDIA Corporation (2008).

[2] 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, in: ISCA’10, 2010, pp. 451–460.

[3] NVIDIA Corporation, CUDA CUBLAS library, PG-00000-002_V3.1, May, 2010.
[4] N. Galoppo, N. K. Govindaraju, M. Henson, D. Manocha, LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware, Proceedings of the 2005 ACM/IEEE SC—05 Conference (SC’05), 2005, pp. 3–14.

[5] V. Volkov, J. Demel, LU, QL and Cholesky factorizations using vector capabilities of gpus, Tech. Rep. UCB/EECS-2008-49, Electrical Engineering and Computer Sciences University of California at Berkeley (2008).

[6] R. Nishtala, R. Vuduc, J. W. Demmel, K. A. Yelick, When cache blocking of sparse matrix multiply works and why, AAECC 18 (2007) 297–311.

[7] E.-J. Im, Optimizing the performance of sparse matrix-vector multiplication, PhD thesis, Rep. No. UCB/CSD-00-1104, University of California, Berkeley May (2000).

[8] K. Kourtis, G. Goumas, N. Koziris, Improving the performance of multi-threaded sparse matrix-vector multiplication using index and value compression, in: Proc. 37th International Conference on Parallel Processing, 2008, pp. 511–519.

[9] M. M. Baskaran, R. Bordwaker, Optimizing sparse matrix-vector multiplication on gpus, Tech. Rep. RC24704(W0812-047), IBM (2008).

[10] J. W. Choi, A. Singh, R. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on gpus, in: PPOPP, 2010, pp. 37–48.

[11] A. Monakov, A. Avetisyan, Implementing blocked sparse matrix-vector multiplication on nvidia gpus, in: Samos 2009, 2009, pp. 289–297.

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

[13] Z. Bai, D. Day, J. Demmel, J. Dongarra, Test matrix collection (non-hermitian eigenvalue problems), release 1, Tech. rep., University of Kentucky, ftp://ftp.ms.uky.edu/pub/misc/bai/Collection (1996).

[14] nVidia, NVIDIA Corporation, NVIDIA CUDA Programming Guide 3.0 (2010).

[15] 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 (3) (2009) 205–223.

[16] AMD, Amd “close to metal” technology unleashes the power of stream computing, Tech. rep., AMD Press Release (2006).
[17] J. Vacata, GPGPU: General Purpose Computation on GPUs, Master’s thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague. (2008).

[18] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically tuning sparse matrix-vector multiplication for gpu architectures, in: HiPEAC 2010, Springer-Verlag Berlin Heidelberg, 2010, pp. 111–125.

[19] Nvidia, Cusp 0.1.1, [http://code.google.com/p/cusp-library/] (2010).

[20] P. Amestoy, T. A. Davis, I. S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (1996) 886–905.