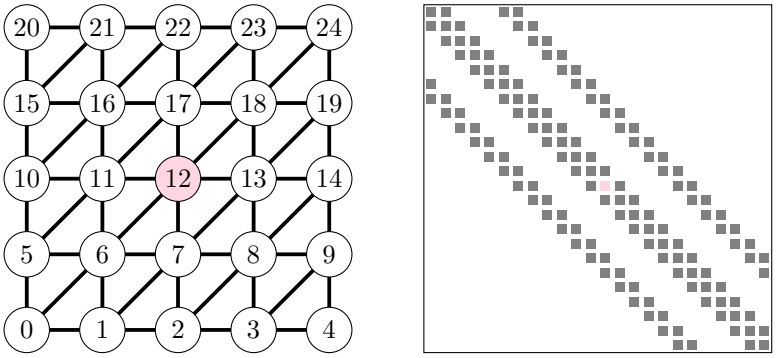
Sparse Matrix-Vector Multiplication with CUDA

Georgii Evtushenko

November 16, 2019

1 Introduction

Standard methods of differential equations discretization usually lead to systems of linear equations. A general feature of produced systems is that the number of entries in each equation depends on local topological features of the discretization. Thus, the matrices generated by these systems contain a lot of zeroes (fig. 1). It's possible to take advantage of knowledge about zeroes' position by storing matrices in special data structures. The abstract data type for these structures is called the sparse matrix. While I was reading about yet another matrix format, I decided to actualize the performance comparison of different matrix formats. This post provides an efficiency review for basic sparse matrix data structures in the context of sparse matrix-vector multiplication (SpMV) on GPU.



(a) Mesh (b) Matrix

Figure 1: A simple finite element mesh model

2 Data Structures for Sparse Matrices

In general, SpMV performance is limited by memory bandwidth. The storage formats used for the sparse matrix define SpMV algorithms. Each of these algorithms has it's own granularity, which impacts performance. The primary distinction among sparse matrix representations is the sparsity pattern, or the structure of the non-zero entries, for which they are best suited. However, I'll start with general sparse matrix formats.

To access the efficiency of SpMV on different sparse matrix formats, I've collected performance data on general matrices from Florida Sparse Matrix Collection. All of the experiments are run on a system with NVIDIA RTX 2080 GPU paired with an Intel Core i7-7700k CPU. Each of the measurements is an average (arithmetic mean) over 30 trials. Before measuring performance, both CPU and GPU frequency were fixed. The speedup is computed by dividing single thread CSR SpMV execution time by GPU one.

2.1 CSR

The *Compressed Sparse Row* (CSR) format is a general sparse matrix format. CSR format consists of three arrays: *row_ptr*, non-zeroes' *columns*, and matrix *values* (fig. 2). The row's non-zero values are stored consequentially in an one-dimensional *values* array. The *row_ptr* array is used to divide *values* array into separate rows. Its size is equal to $n\_rows + 1$. The last entry in *row_ptr* stores number of non-zeroes (NNZ) in the matrix. That allows fast querying of non-zeroes number in a particular row ($row\_ptr[row + 1] - row\_ptr[row]$). For each non-zero value column index is stored in *columns* array.

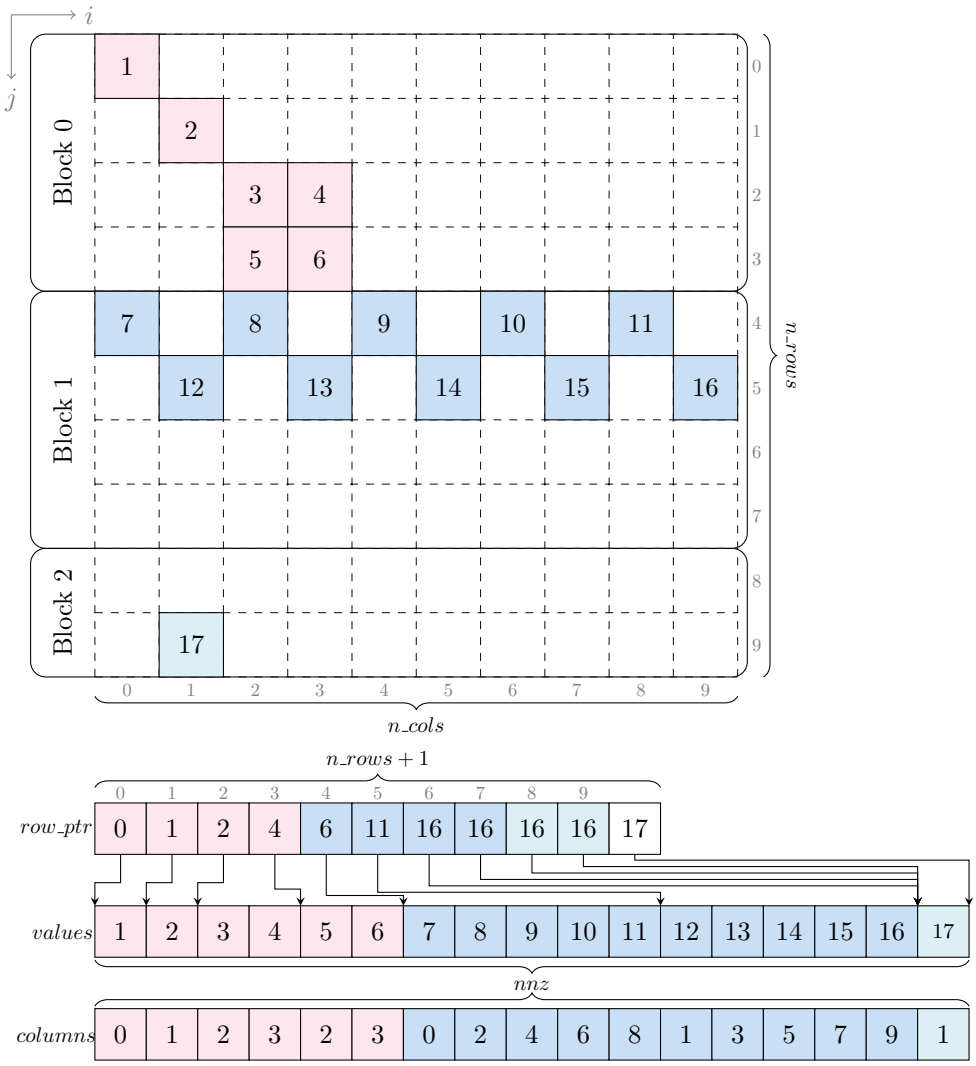


Figure 2: Example of Compressed Sparse Row (CSR) matrix format

Let's assume for simplicity that there are four threads in each CUDA thread block. General CSR SpMV implementation works at the granularity of threads per row (fig. 3). Hence, the matrix in figure 2 is processed by three thread blocks. This implementation is usually referenced as CSR-Scalar (list. 1).

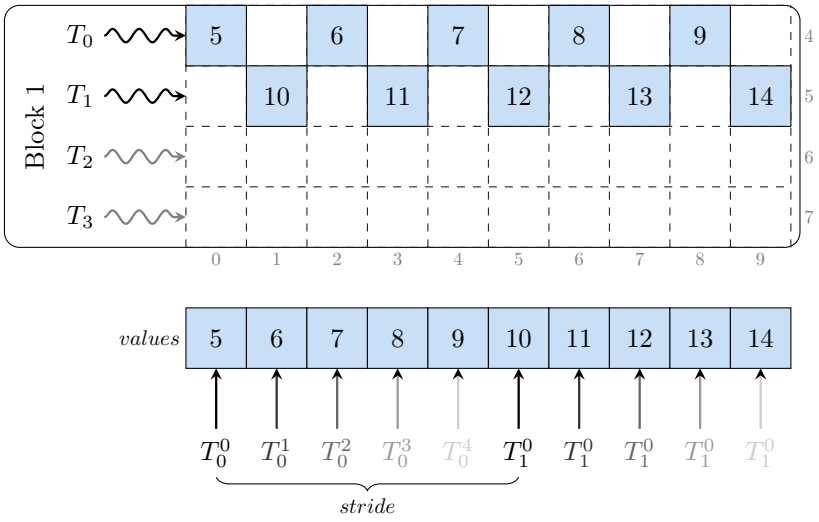


Figure 3: CSR-Scalar block's threads' work distribution

```
template <typename data_type>
__global__ void csr_spmv_kernel (
    unsigned int n_rows,
    const unsigned int *col_ids,
    const unsigned int *row_ptr,
    const data_type *data,
    const data_type *x,
    data_type *y)
{
  unsigned int row = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < n_rows)
  {
    const int row_start = row_ptr[row];
    const int row_end = row_ptr[row + 1];

    data_type sum = 0;
    for (unsigned int element = row_start; element < row_end; element++)
      sum += data[element] * x[col_ids[element]];
    y[row] = sum;
  }
}
```

Listing 1: Naive SpMV kernel for the CSR-Scalar sparse matrix format

Presented implementation of CSR SpMV algorithm on GPU is usually considered very inefficient. The reasons for inefficiency are load balancing, thread divergence, and memory access pattern. As shown in figure 3, only half of the block threads have non-zeroes to process. Thus, a single dense row can arbitrarily delay the execution while all other cores are idle. Moreover, as shown in figure 3, adjacent threads access matrix values in a strided way. When concurrent threads simultaneously access memory addresses that are far apart in physical memory, then there is no chance for the hardware to combine the accesses. Performance results for naive CSR-Scalar implementation are presented in table 1.

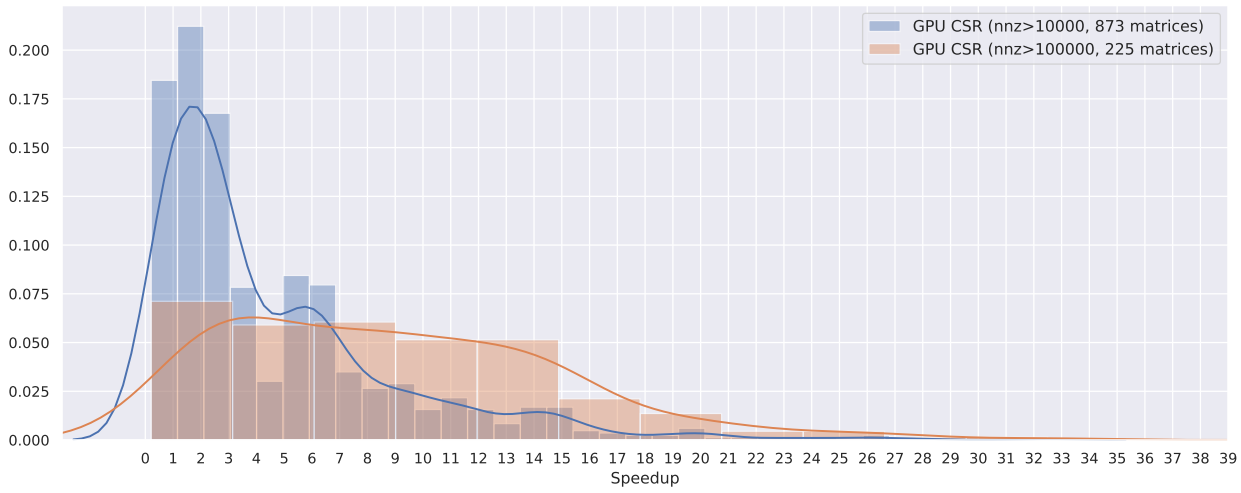
	float		double	
NNZ lower limit	avg	max	avg	max
10000	4.57	32.50	3.78	29.47
100000	8.90	32.50	7.24	29.47

Table 1: CSR-Scalar speedup

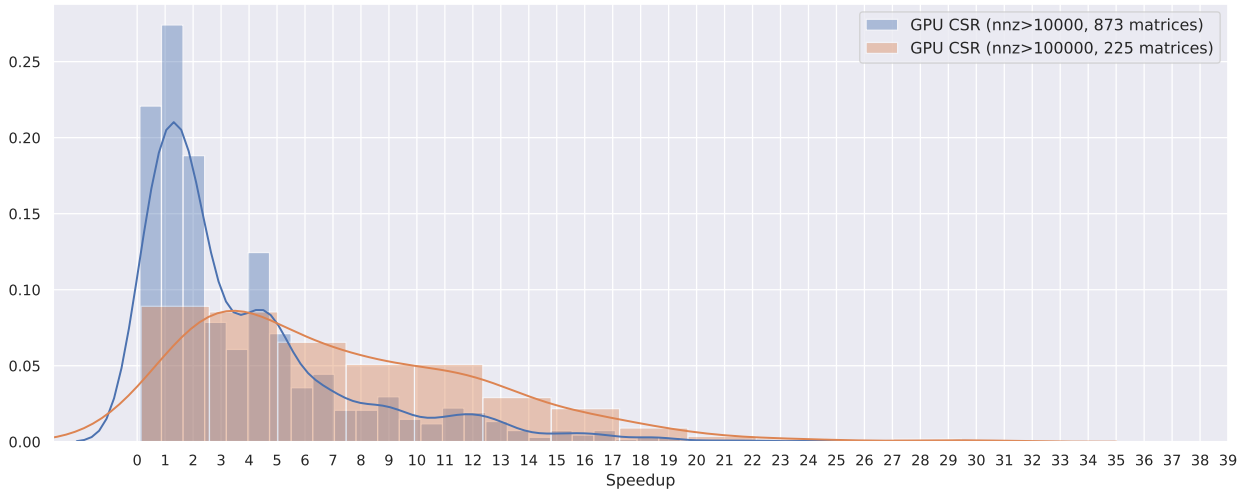
The speedup distribution is shown in figures 4a and 4b. To answer the question how naive described implementation really is I've compared it with the NVIDIA CUDA Sparse Matrix library (cuSPARSE) CSR implementation (tab. 2), which has a better average speedup (fig. 4c and 4d).

	float		double	
NNZ lower limit	avg	max	avg	max
10000	5.69	31.44	4.68	25.42
100000	13.62	31.44	10.65	25.42

Table 2: CSR (cuSPARSE) speedup

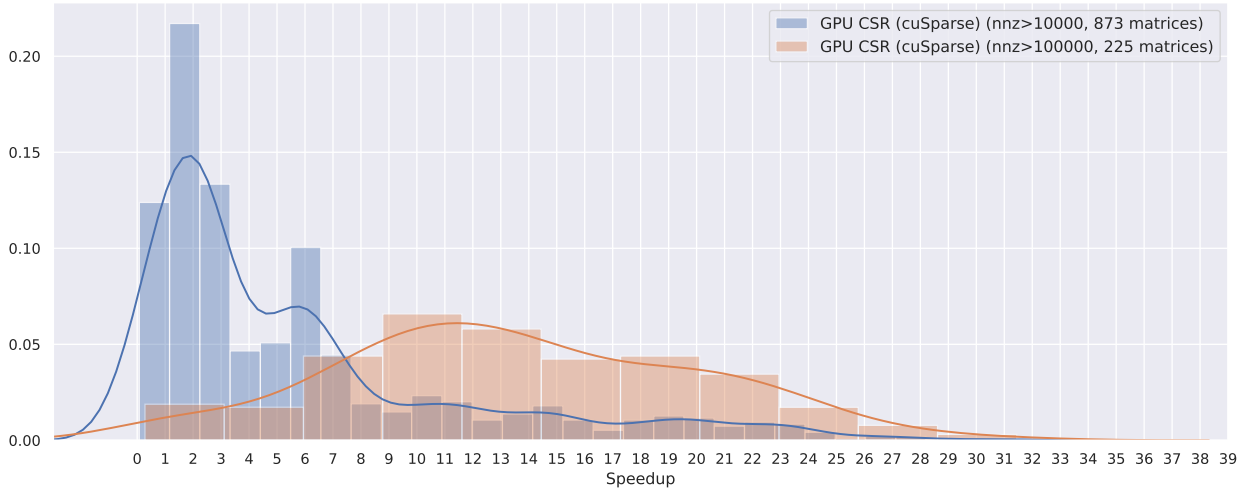


(a) CSR-Scalar speedup (float)

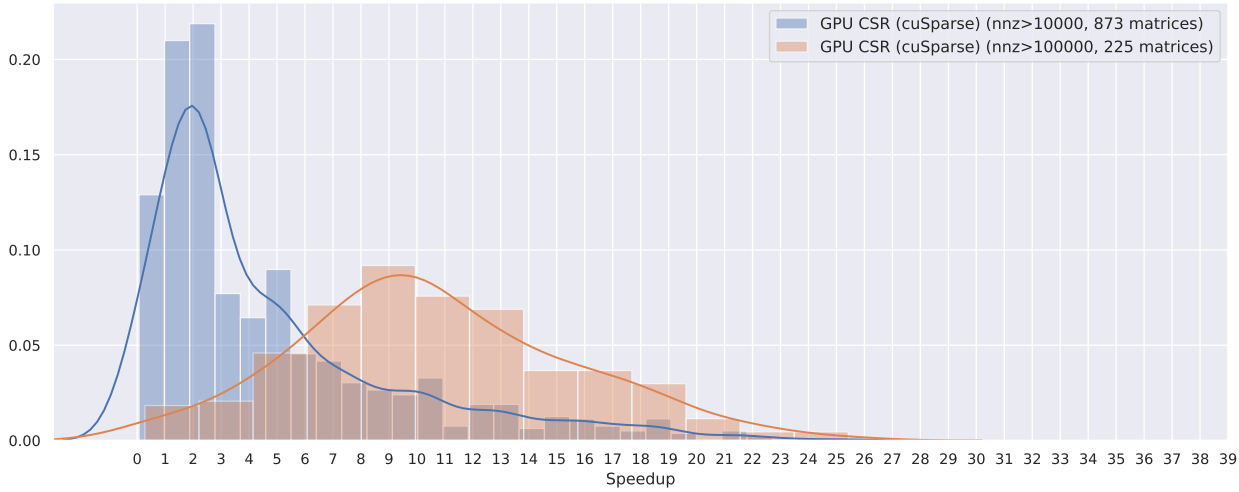


(b) CSR-Scalar speedup (double)

These results show that there is room for optimization of CSR SpMV. The first possible optimization is to assign warp per row instead of thread. This algorithm (list. 3) is called CSR-Vector. The vector kernel accesses indices and data contiguously (fig. 4), and therefore overcomes the principal deficiency of the scalar approach. Unlike the previous CSR implementation, which used one thread per matrix row, this optimization requires coordination among threads within the same warp.



(c) CSR cuSPARSE speedup (float)



(d) CSR cuSPARSE speedup (double)

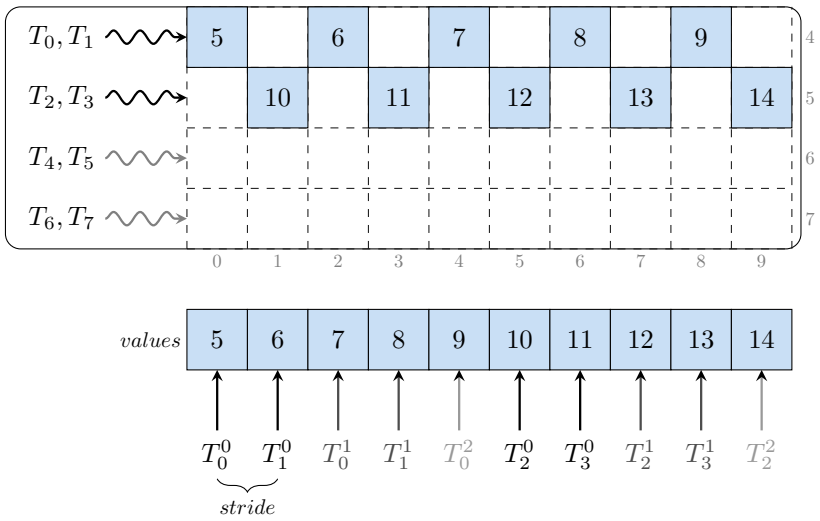


Figure 4: CSR-Scalar block's threads' work distribution

In the case of CSR-Vector reduction might be implemented using warp-level primitives (list. 2). In that case, the data exchange is performed between registers and more efficient than going through shared memory, which requires a load, a store, and an extra register to hold the address.

```
template <class T>
__device__ T warp_reduce (T val)
{
  for (int offset = warpSize / 2; offset > 0; offset /= 2)
    val += __shfl_down_sync (FULL_WARP_MASK, val, offset);

  return val;
}
```

Listing 2: Warp reduction

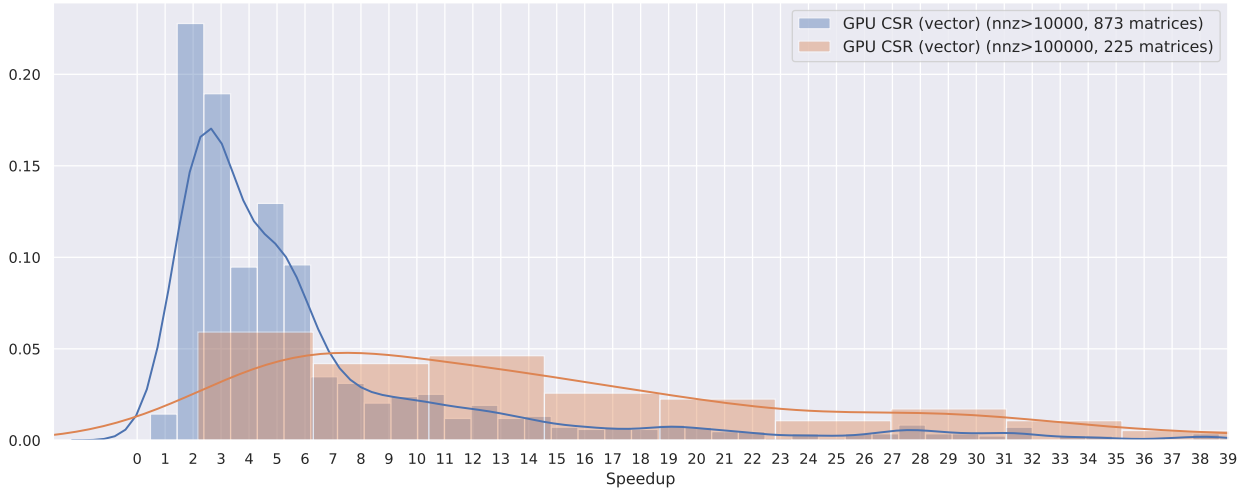
```
template <typename data_type>
__global__ void csr_spmv_vector_kernel (
    unsigned int n_rows,
    const unsigned int *col_ids,
    const unsigned int *row_ptr,
    const data_type *data,
    const data_type *x,
    data_type *y)
{
  const unsigned int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
  const unsigned int warp_id = thread_id / 32;
  const unsigned int lane = thread_id % 32;

  const unsigned int row = warp_id; ///< One warp per row

  data_type sum = 0;
  if (row < n_rows)
  {
    const unsigned int row_start = row_ptr[row];
    const unsigned int row_end = row_ptr[row + 1];

    for (unsigned int element = row_start + lane; element < row_end; element += 32)
      sum += data[element] * x[col_ids[element]];
  }

  sum = warp_reduce (sum);

  if (lane == 0 && row < n_rows)
    y[row] = sum;
}
```

Listing 3: SpMV kernel for the CSR sparse matrix format (vector)

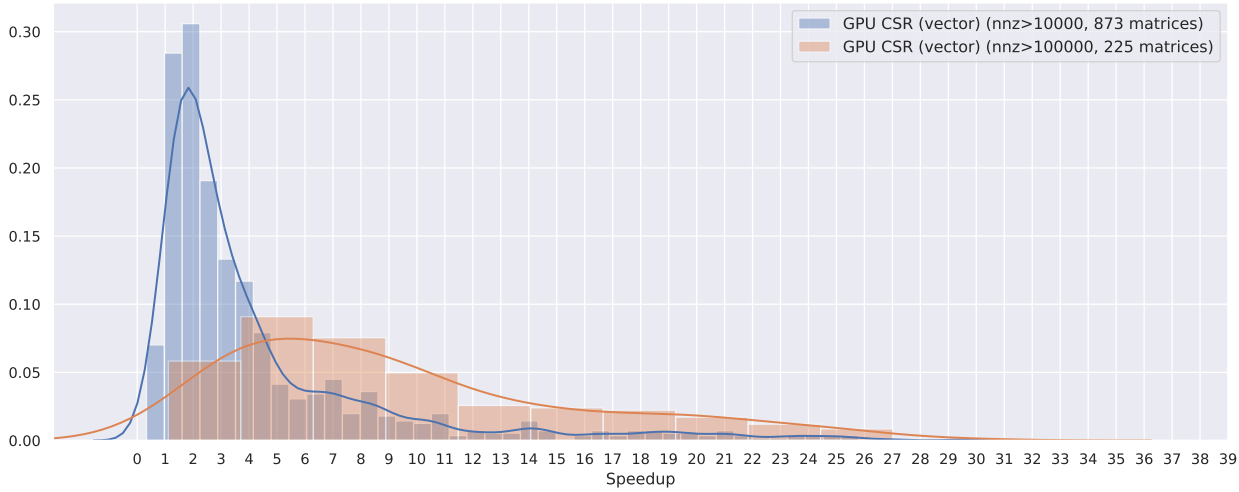
CSR-Vector has better speedup (tab. 4) and speedup distribution (fig. 5a and 5b) than CSR-Scalar (for both float and double matrices) and cuSPARSE implementation (for float matrices).

	float		double	
NNZ lower limit	avg	max	avg	max
10000	6.60	43.46	4.37	29.62
100000	14.40	43.46	9.50	29.62

Table 3: CSR-Vector speedup

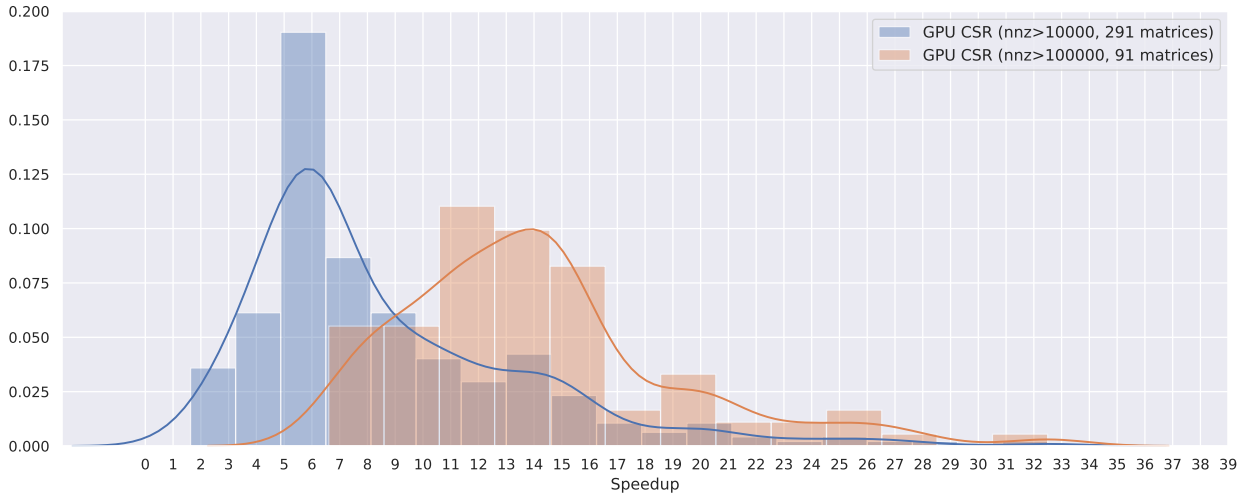


(a) CSR-Vector speedup (float)

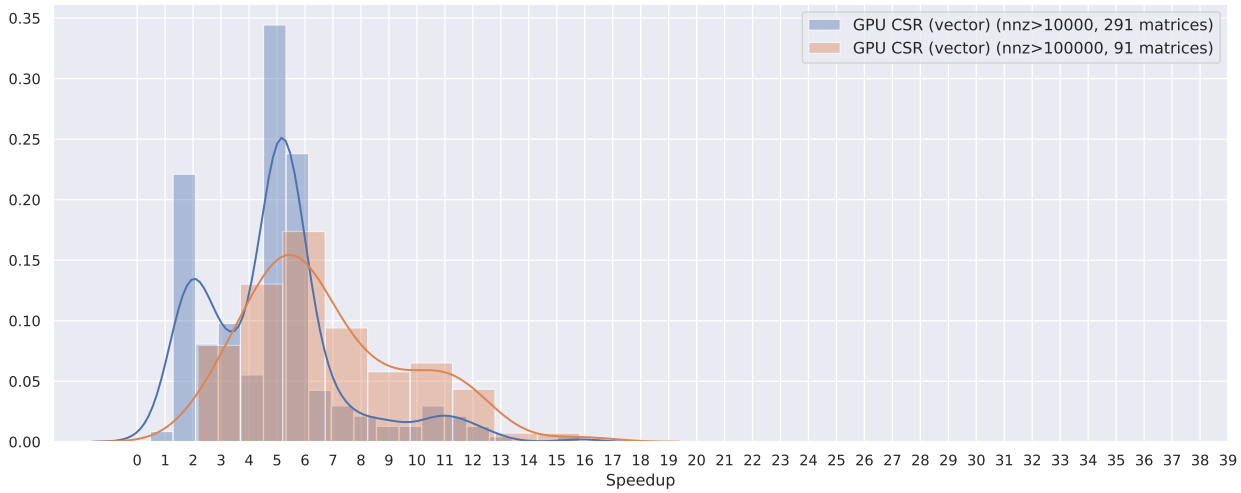


(b) CSR-Vector speedup (double)

However, CSR-Scalar outperforms CSR-Vector on about 33% of float matrices with 10000 nnz lower limit and on 40% of float matrices with 100000 nnz lower limit (fig 5c and 5d). On that matrices, CSR shows average speedup equal to 8.57 while CSR-Vector only 4.80.



(c) CSR-Vector speedup (float)



(d) CSR-Vector speedup (double)

To discover further improvements of CSR SpMV implementation, we need to consider the first matrix part from figure 2. In the first four rows of the matrix, there is only one non-zero value per row. In that case whole warp's threads except first are idle. In this case, it's possible for naive CSR SpMV implementation to outperform vector implementation. There is an SpMV algorithm for the CSR matrix format that doesn't depend on nnz/row ratio. The CSR-Adaptive changes it's behavior depending on the nnz in each row (list. 4). After selecting non-zeroes per block value, additional array (row_blocks) for storing block rows is constructed. If some rows contain small nnz, they'll be gathered into one block. Then CUDA threads block is assigned to each rows block. The case of multiple rows in one rows block is called CSR-Stream. If there is only one row in rows block, the CSR-Vector will be called. If this row exceeds nnz_per_wg than CSR-VectorL variant will be used. The main difference between CSR-Vector and CSR-VectorL is that CSR-VectorL allows executing multiple CSR-VectorL on one row and then reducing the results by using atomic operations.

```
1   template <typename data_type>
2   __global__ void csr_adaptive_spmv_kernel (
3       const unsigned int n_rows,
4       const unsigned int *col_ids,
5       const unsigned int *row_ptr,
6       const unsigned int *row_blocks,
7       const data_type *data,
8       const data_type *x,
9       data_type *y)
10  {
11    const unsigned int block_row_begin = row_blocks[blockIdx.x];
12    const unsigned int block_row_end = row_blocks[blockIdx.x + 1];
13
14    __shared__ data_type cache[NNZ_PER_WG];
15
16    if (block_row_end - block_row_begin > 1)
17    {
18      /// CSR-Stream case...

94    }
95    else
96    {
97      const unsigned int nnz = row_ptr[block_row_end] - row_ptr[block_row_begin];
98
99      if (nnz <= 64)
100     {
101       /// CSR-Vector case...

118     }
119     else
120     {
121       /// CSR-VectorL case...

151     }
152   }
153 }
```

Listing 4: SpMV kernel for the CSR-Adaptive sparse matrix format

The CSR-Vector and CSR-VectorL parts are quite similar CSR-Vector algorithm, so I won't include listing here. Figure 5 illustrates memory access pattern of the CSR-Stream part. It stores partial sums in shared memory of GPU and then reduces them. The partial results in cache in figure 5 are calculated with x filled with 1. The source code of CSR-Stream is presented in listing 5.

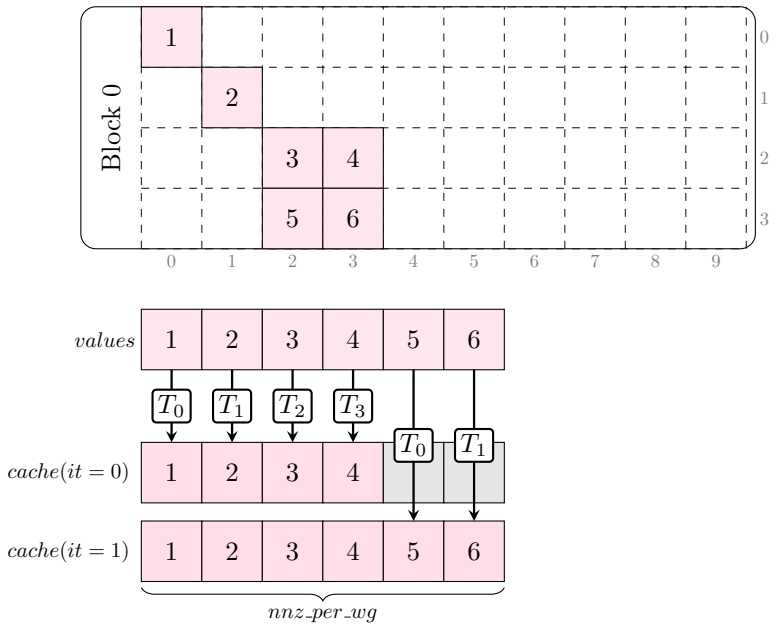


Figure 5: CSR-Stream memory access pattern

```
19  const unsigned int i = threadIdx.x;
20  const unsigned int block_data_begin = row_ptr[block_row_begin];
21  const unsigned int thread_data_begin = block_data_begin + i;
22
23  if (i < nnz)
24    cache[i] = data[thread_data_begin] * x[col_ids[thread_data_begin]];
25  __syncthreads ();
26
27  const unsigned int threads_for_reduction = prev_power_of_2 (blockDim.x / (block_row_end - block_row_begin));
28
29  if (threads_for_reduction > 1)
30    {
31      /// Reduce all non zeroes of row by multiple thread
32      const unsigned int thread_in_block = i % threads_for_reduction;
33      const unsigned int local_row = block_row_begin + i / threads_for_reduction;
34
35      data_type sum = 0.0;
36
37      if (local_row < block_row_end)
38        {
39          const unsigned int local_first_element = row_ptr[local_row] - row_ptr[block_row_begin];
40          const unsigned int local_last_element = row_ptr[local_row + 1] - row_ptr[block_row_begin];
41
42          for (unsigned int local_element = local_first_element + thread_in_block;
43               local_element < local_last_element;
44               local_element += threads_for_reduction)
45            {
46              sum += cache[local_element];
47            }
48        }
49      __syncthreads ();
50      cache[i] = sum;
51
52      /// Now each row has threads_for_reduction values in cache
53      for (int j = threads_for_reduction / 2; j > 0; j /= 2)
54        {
55          /// Reduce for each row
56          __syncthreads ();
57
58          const bool use_result = thread_in_block < j && i + j < NNZ_PER_WG;
59
60          if (use_result)
61            sum += cache[i + j];
62          __syncthreads ();
63
64          if (use_result)
65            cache[i] = sum;
66        }
67
68      if (thread_in_block == 0 && local_row < block_row_end)
69        y[local_row] = sum;
70    }
71  else
72    {
73      /// Reduce all non zeroes of row by single thread
74      unsigned int local_row = block_row_begin + i;
75      while (local_row < block_row_end)
76        {
77          data_type sum = 0.0;
78
79          for (unsigned int j = row_ptr[local_row] - block_data_begin;
80               j < row_ptr[local_row + 1] - block_data_begin;
81               j++)
82            {
83              sum += cache[j];
84            }
85
86          y[local_row] = sum;
87          local_row += NNZ_PER_WG;
88        }
89    }
```

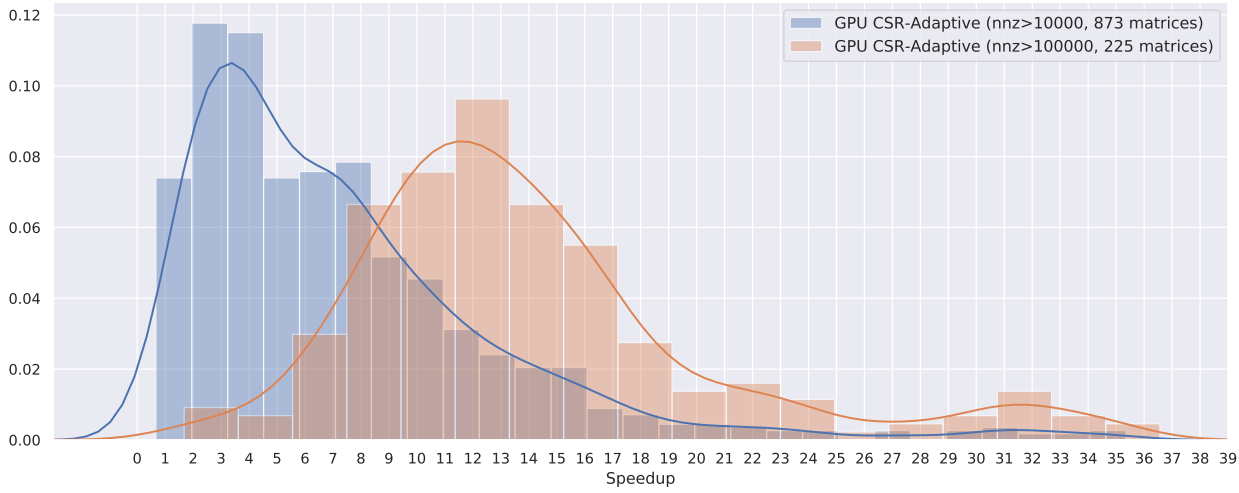
Listing 5: CSR-Stream implementation

On the discussed set of matrices (fig. 5c and 5d), where CSR outperformed CSR-Vector, CSR-Adaptive

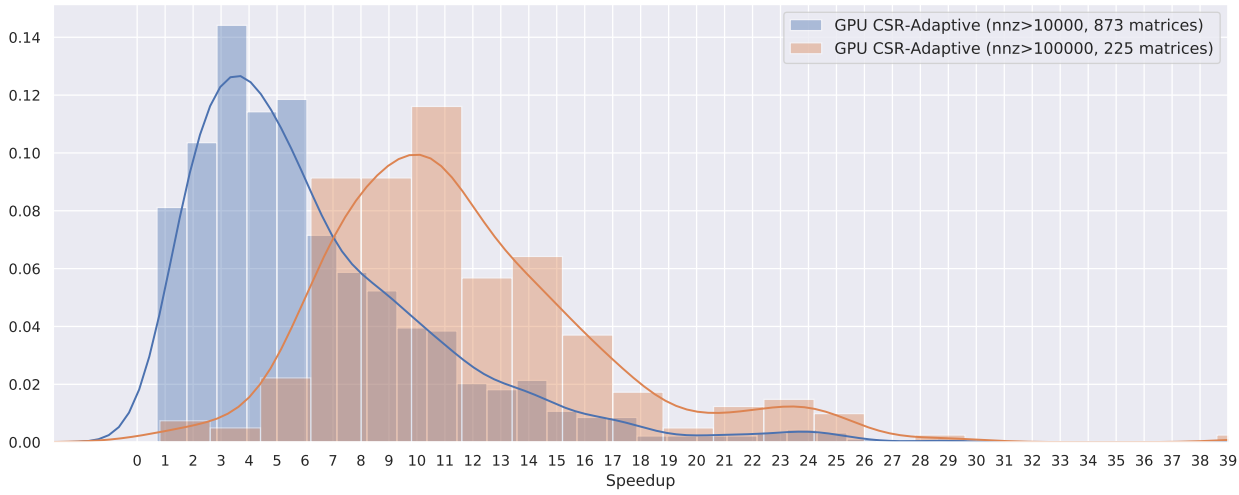
shows better speedup. CSR-Adaptive outperforms CSR-Scalar on those 291 matrices. Although CSR-Adaptive might be outperformed by CSR-Vector on some long-row matrices, it has better speedup in average (tab. 4, fig. 6a and 6b). The main advantage of CSR-Adaptive is that you won't need to change the code that generates a matrix if your code already uses CSR. The matrix formats presented below don't have this quality.

	float		double	
NNZ lower limit	avg	max	avg	max
10000	7.37	48.19	6.39	40.39
100000	14.27	48.19	11.72	40.39

Table 4: CSR-Adaptive speedup



(a) CSR-Adaptive speedup (float)



(b) CSR-Adaptive speedup (double)

2.2 ELL

The problem of noncoalesced memory accesses of CSR can be addressed by applying data padding and transposition on the sparse matrix data (fig. 6). The Ellpack-Itpack (ELL) sparse matrix format assumes that each row contains at most *elements_in_rows* elements and *elements_in_rows* is small. All rows are zero-padded to that value. Unlike CSR, the rows pointers array is of no need. ELL is most efficient when the maximum number of nonzeros per row does not substantially differ from the average.

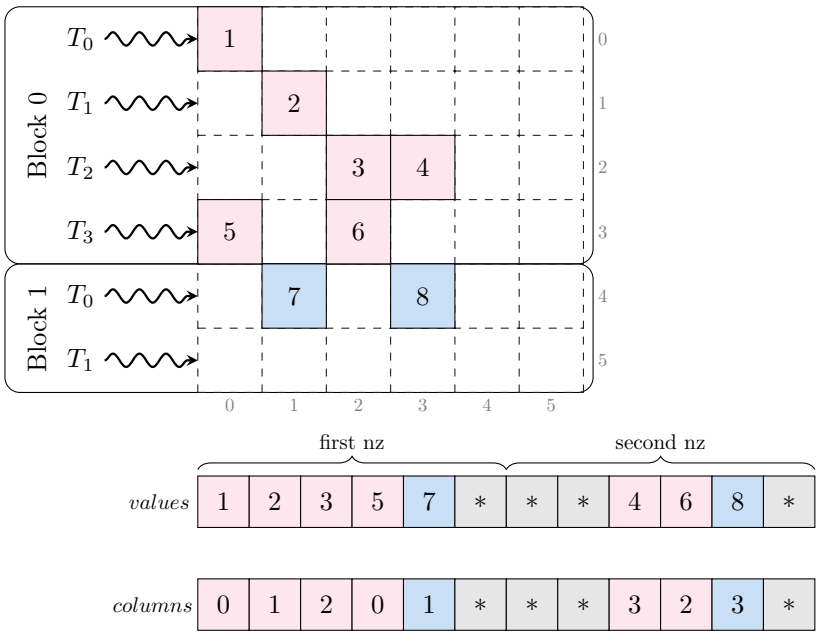


Figure 6: Example of ELL matrix format

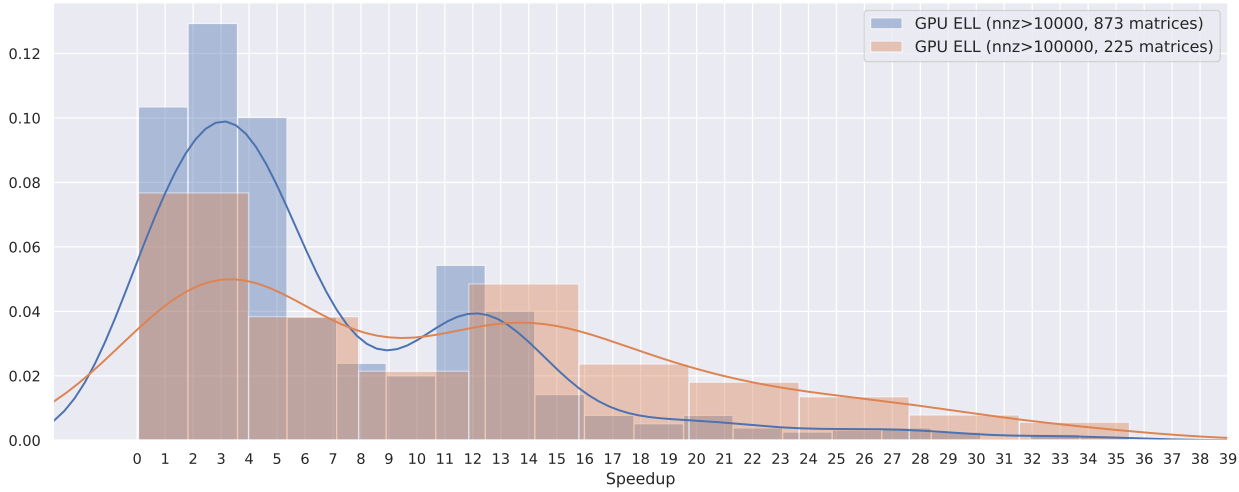
Kernel for ELL matrix format is presented in the listing 6. With element padding of the ELL format it's easy to get next row's element position by simply adding the number of rows in the matrix. The padding also fixes the number of iteration for each thread, so there is no control flow divergence in warps. Elimination of control flow divergence and enabling of memory coalescing allow ELL SpMV kernel to outperform CSR-Scalar implementation on many matrices (tab. 5, fig. 7a and 7b).

```
template <typename data_type>
__global__ void ell_spmv_kernel (
    unsigned int n_rows,
    unsigned int elements_in_rows,
    const unsigned int *col_ids,
    const data_type*data,
    const data_type*x,
    data_type*y)
{
  unsigned int row = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < n_rows)
  {
    data_type sum = 0;
    for (unsigned int element = 0; element < elements_in_rows; element++)
    {
      const unsigned int element_offset = row + element * n_rows;
      sum += data[element_offset] * x[col_ids[element_offset]];
    }
    y[row] = sum;
  }
}
```

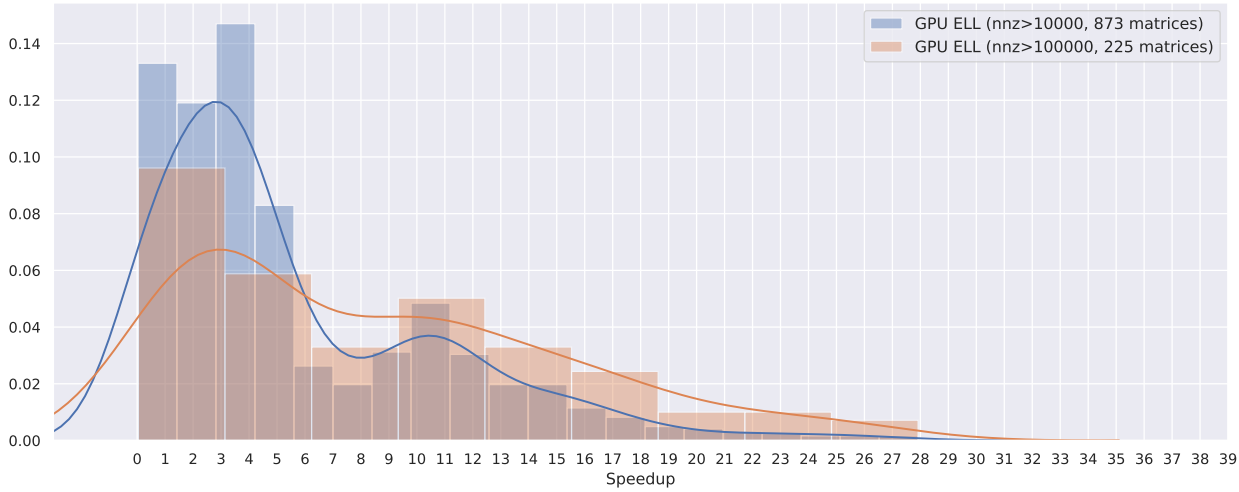
Listing 6: ELL implementation

	float		double	
NNZ lower limit	avg	max	avg	max
10000	6.73	35.47	5.57	27.92
100000	11.16	35.47	8.44	27.92

Table 5: ELL speedup



(a) ELL speedup (float)



(b) ELL speedup (double)

The obvious disadvantage of ELL format consist in padding itself. In case of matrix with a few long rows, ELL format will result in excessive number of padded elements. There are a lot of matrices in Florida Collection, that coudn't fit into 8GB of my GPU because of ELL's padding. In some cases it leads to the situation, where CSR-Scalar outperform ELL implementation. To eliminate this issue, it's possible to remove long rows' extra nnz from ELL matrix into different matrix. It's important to note that extracted matrix would have unordered scheme. Many rows will likely be missing from that scheme, so CSR using would be inefficient. One of the formats that could handle that case is COO.

2.3 COO

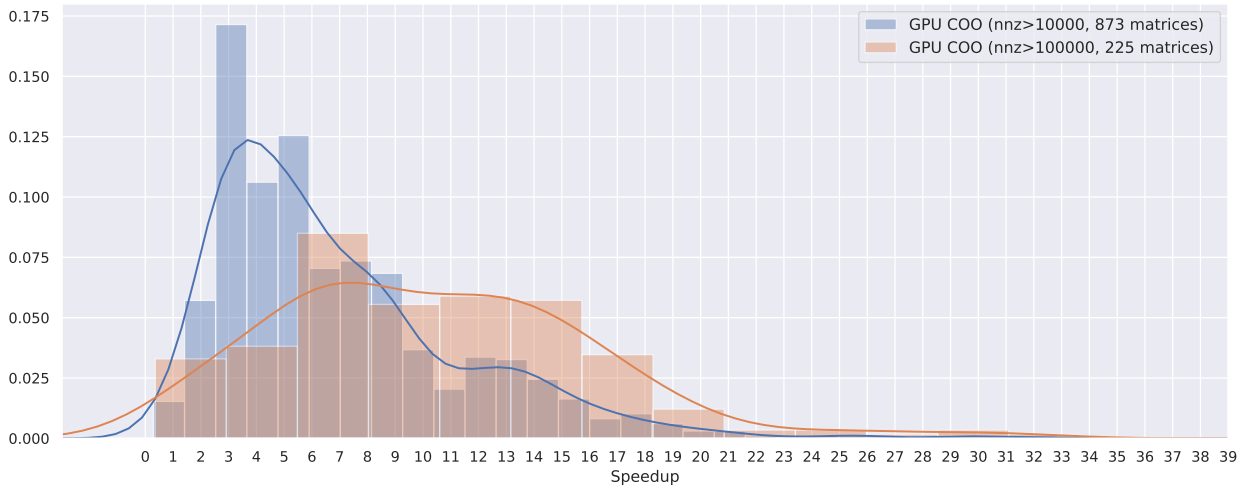
The coordinate (COO) matrix format is a the simpliest one. For each NZ it stores it's column and row indicies. Therefore, COO doesn't map elements in rows. That leads us to the necessity of atomic operations in COO kernel (list 7).

```cpp
template <typename data_type>
__global__ void coo_spmv_kernel (
    unsigned int n_elements,
    const unsigned int *col_ids,
    const unsigned int *row_ids,
    const data_type *data,
    const data_type *x,
    data_type *y)
{
  unsigned int element = blockIdx.x * blockDim.x + threadIdx.x;

  if (element < n_elements)
    atomicAdd (y + row_ids[element], data[element] * x[col_ids[element]]);
}
```

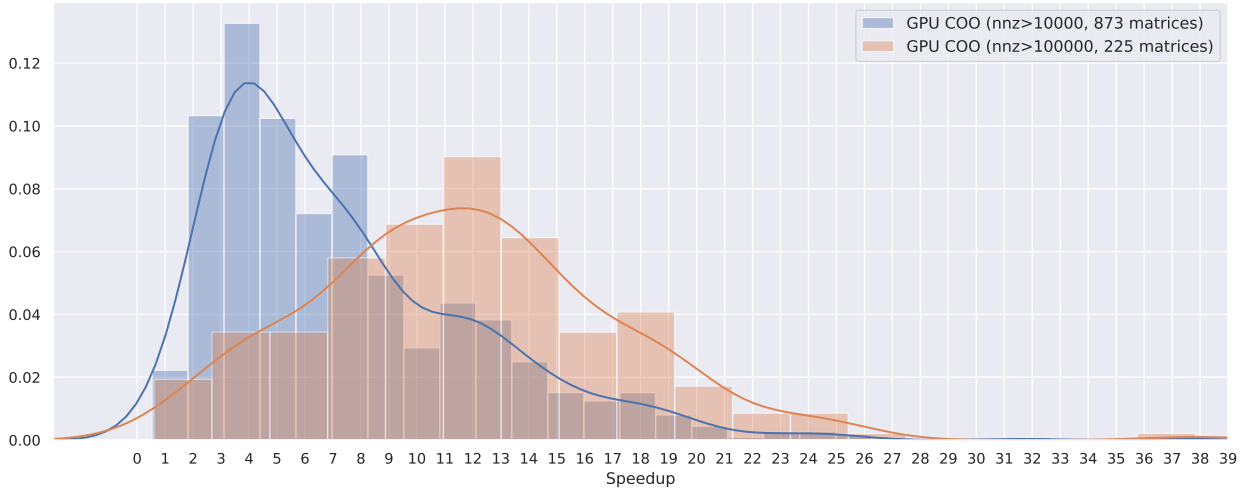
Listing 7: COO implementation

	float		double	
NNZ lower limit	avg	max	avg	max
10000	6.96	54.15	7.37	37.83
100000	10.55	54.15	11.69	37.83

Table 6: COO speedup



(c) COO speedup (float)



(d) COO speedup (double)

COO SpMV implementation works at the granularity of threads per element (7). Atomic updates to the result vector reduce performance. The wider rows in COO format, the more serialized SpMV is. This fact can be noticed from figure 7. To improve performance of this format it's possible to slice the matrix info chunks with the rows count that fits into shared memory.

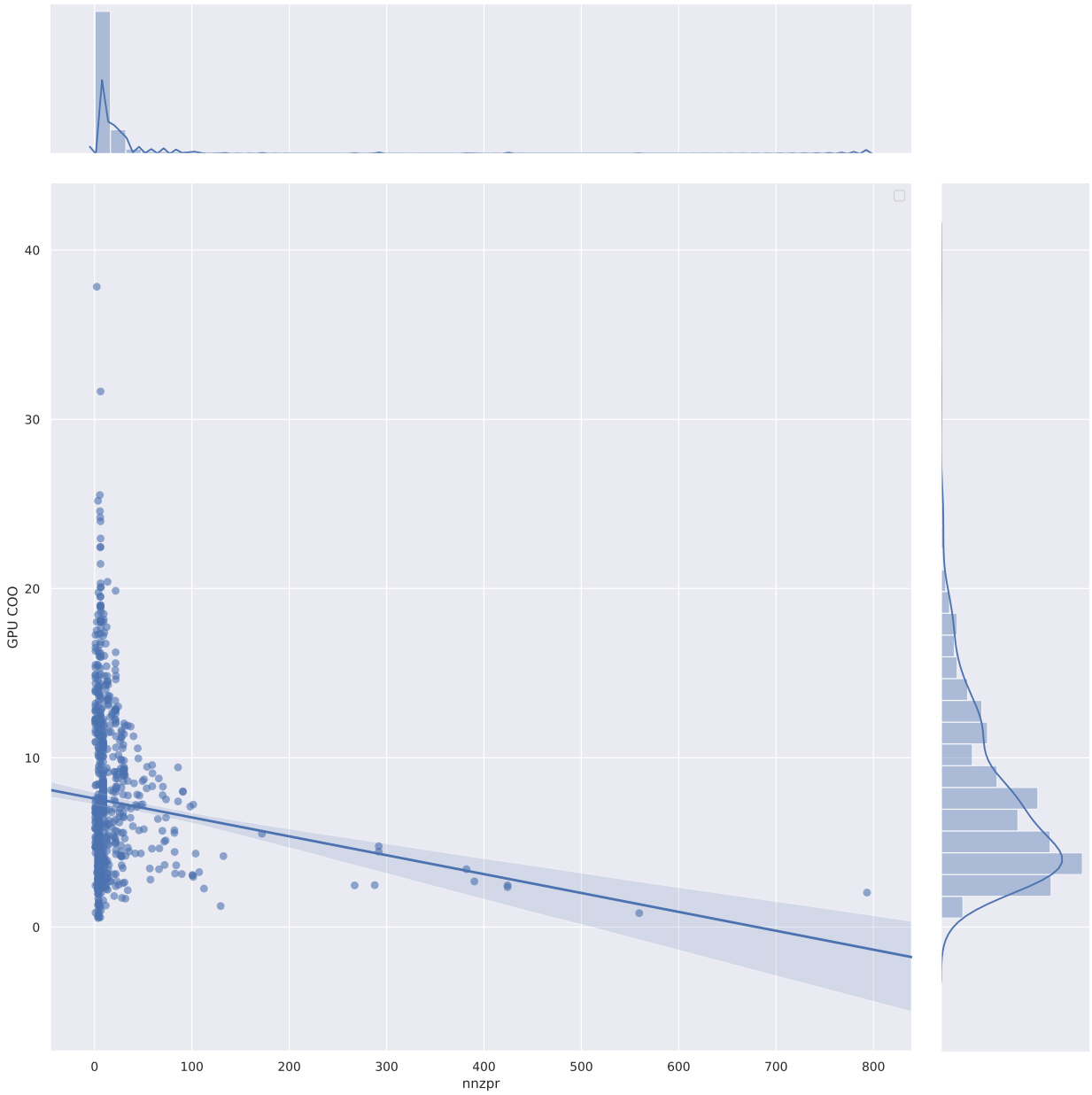


Figure 7: The dependence of the COO parameter on the average NNZ

Matrix format that uses shared memory to improve atomic operations performance in COO SpMV is called Sliced COO (SCOO). To reduce shared memory bank conflicts, SCOO allows multiple lanes in the shared memory for updating the intermediate results of a single row. Reducing slice size increases lanes size and thus more shared memory lanes are available.

	float		double	
NNZ lower limit	avg	max	avg	max
10000	6.82	38.63	4.60	26.78
100000	12.46	38.63	7.43	26.78

Table 7: SCOO speedup

2.4 Hybrid

It's possible to use ELL matrix format on regular part of the matrix and COO on the elements removed from extra-long rows. This scheme significantly reduces the number of padded elements in ELL format. This approach is often called as hybrid. There is different options for combining results of ELL and COO SpMV. In this post I use atomic case (list. 8).

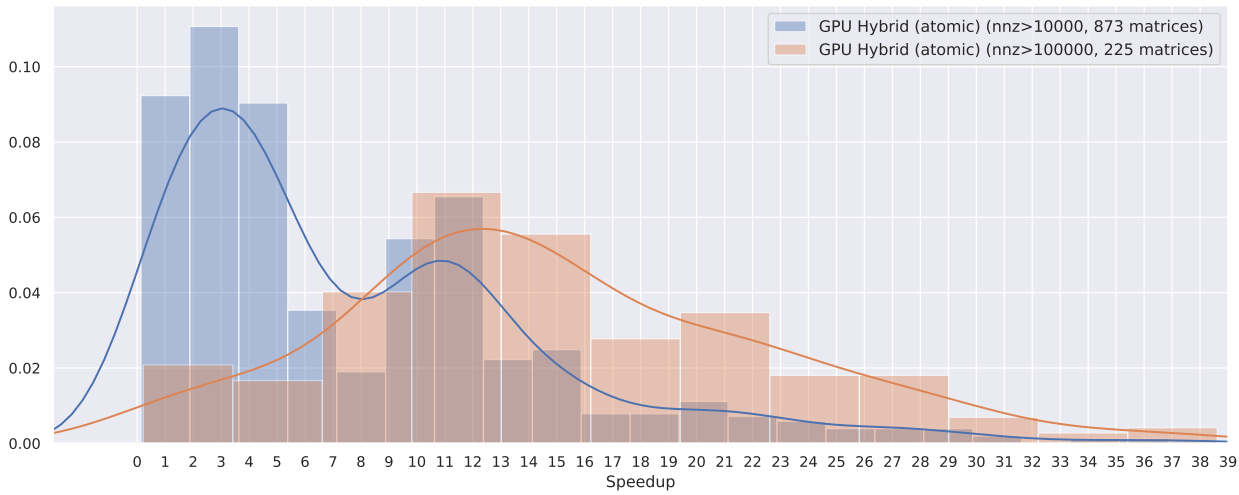
```
template <typename data_type>
__global__ void hybrid_spmv_kernel (
    unsigned int n_rows,
    unsigned int n_elements,
    unsigned int elements_in_rows,
    const unsigned int *ell_col_ids,
    const unsigned int *col_ids,
    const unsigned int *row_ids,
    const data_type *ell_data,
    const data_type *coo_data,
    const data_type *x,
    data_type *y)
{
  const unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx < n_rows)
  {
    const unsigned int row = idx;

    data_type sum = 0;
    for (unsigned int element = 0; element < elements_in_rows; element++)
    {
      const unsigned int element_offset = row + element * n_rows;
      sum += ell_data[element_offset] * x[ell_col_ids[element_offset]];
    }
    atomicAdd (y + row, sum);
  }

  for (unsigned int element = idx; element < n_elements; element += blockDim.x * gridDim.x)
  {
    const data_type sum = coo_data[element] * x[col_ids[element]];
    atomicAdd (y + row_ids[element], sum);
  }
}
```

Listing 8: Hybrid implementation

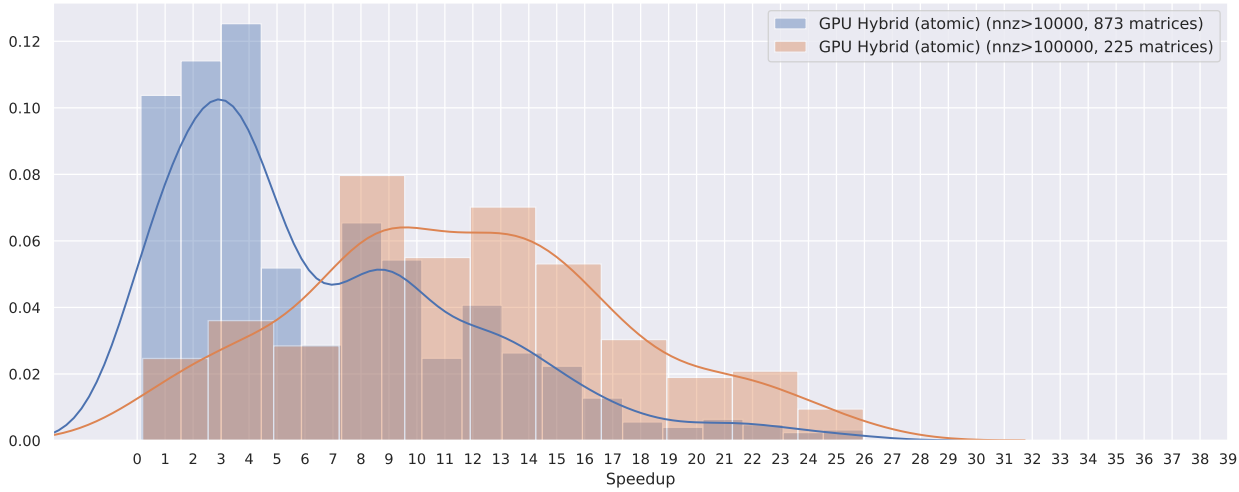
Althought, the average performance results (tab. 8, fig. 8a and 8b) are quite close to CSR-Adaptive SpMV, Hybrid format requires extra actions of splitting matrix, which might require rewriting of matrix calculation code base.

	float		double	
NNZ lower limit	avg	max	avg	max
10000	7.73	38.62	6.51	25.96
100000	14.92	38.62	11.59	25.96

Table 8: HYB speedup



(a) HYB speedup (float)



(b) HYB speedup (double)

3 Conclusion

To conclude this post I would like to show you some misleading results. I've selected some matrices (tab. 9) to show the obvious fact that there is no silver bullet. The leader changes even after data type change (fig. 8c and 8d). In my next post I'm going to focuse on block matrix formats generated by real applications. Source code and pdf version of this post are available in github.

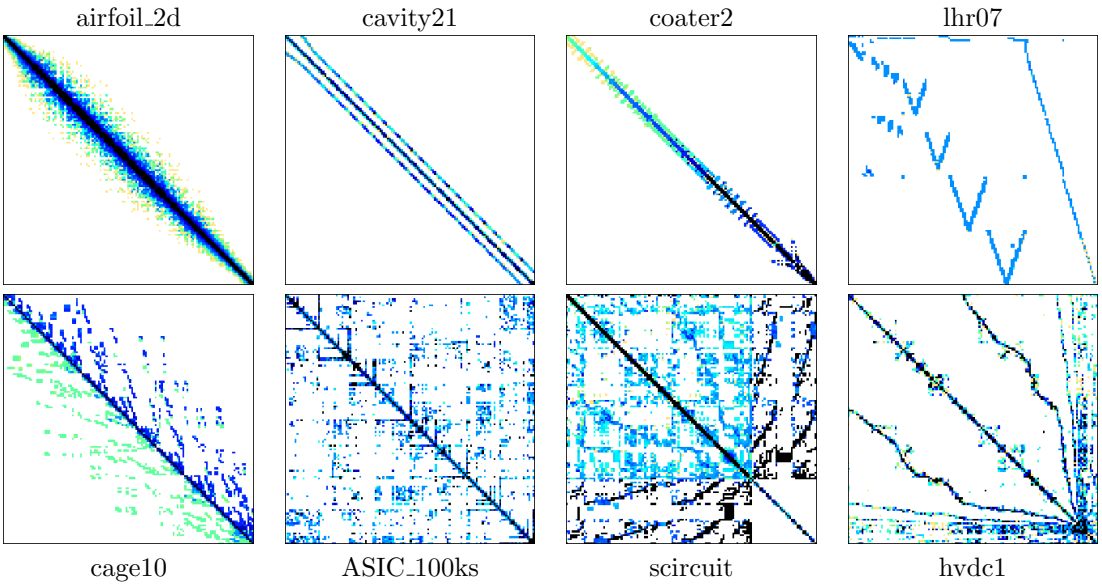
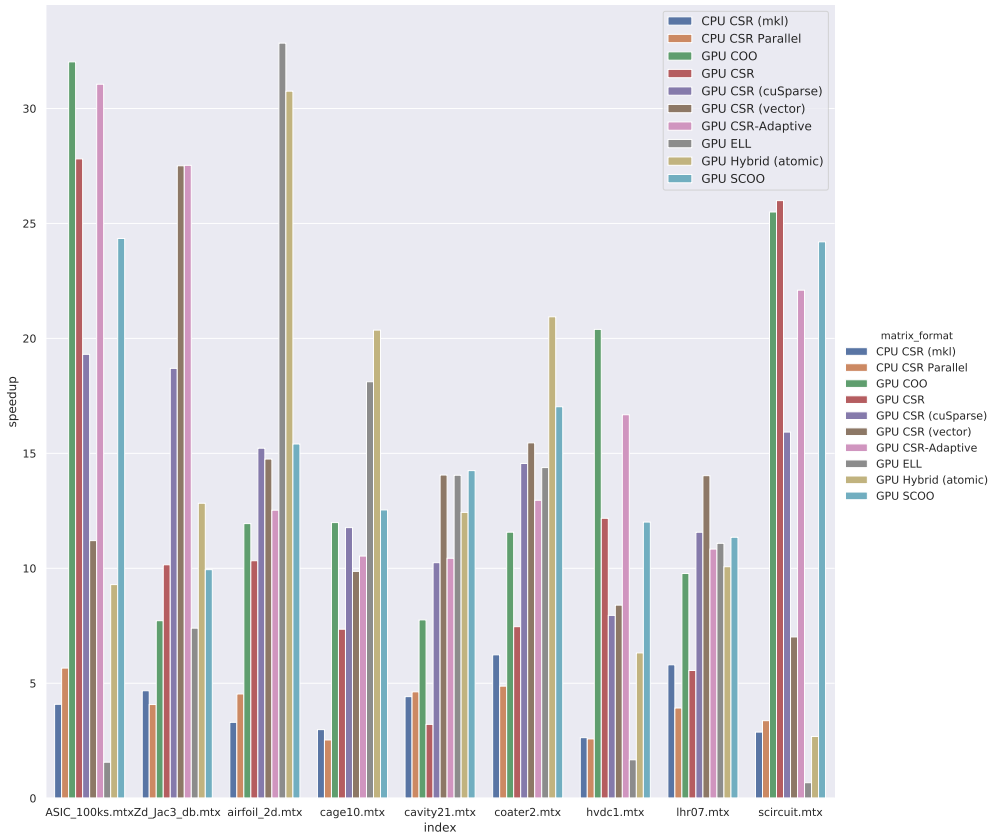
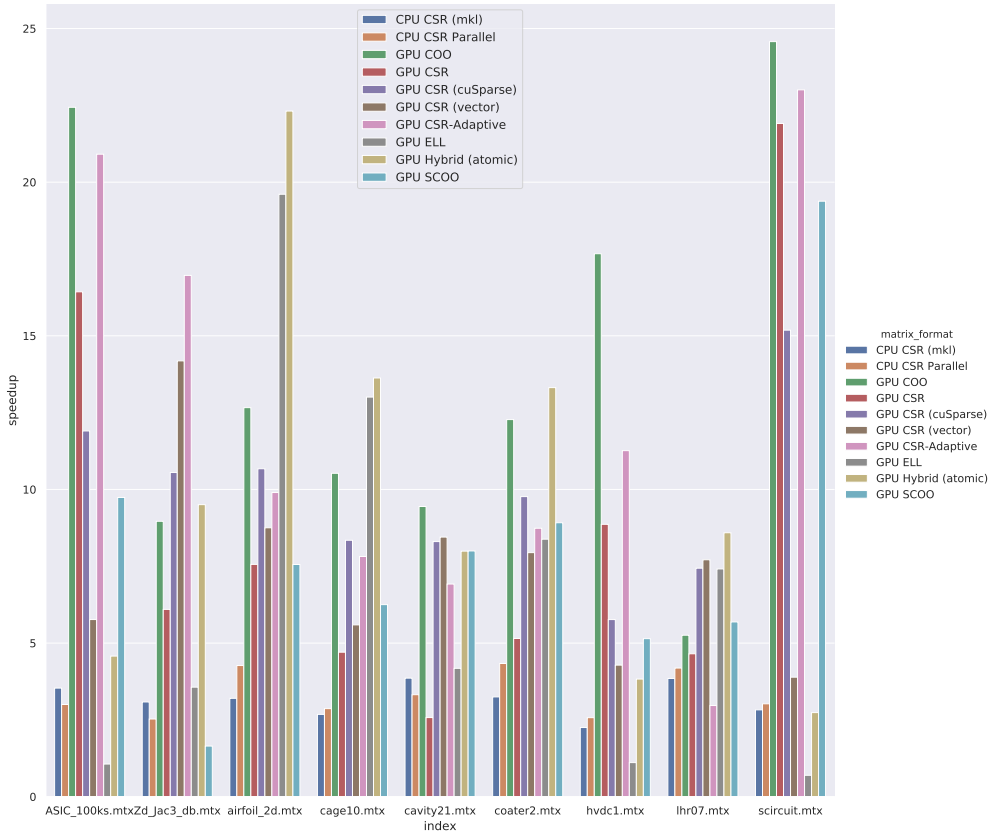


Table 9: Structure of the selected matrices



(c) Speedup for the selected matrices (float)



(d) Speedup for the selected matrices (double)