Efficient Sparse Matrix-Vector Multiplication on CUDA

Federico Villa

Politecnico di Milano

federico5.villa@mail.polimi.it

Abstract

Sparse Matrix-Vector Multiplications are used for many scientific computations, such as graphics processing, graph algorithms, numerical analysis, conjugate gradients… This linear algebra problem is a great example of a problem that can be solved much more efficiently with the use of GPU and task parallelization, instead of CPU.

By switching from the latency-oriented model of the CPU to the throughput oriented model of the GPU, it is possible to increase the performance of the operation by more than 200x.

What is a Sparse Matrix?

A sparse matrix is a matrix whose elements are mainly zeros. It’s the opposite of a dense matrix, in which most of the elements are non-zero.

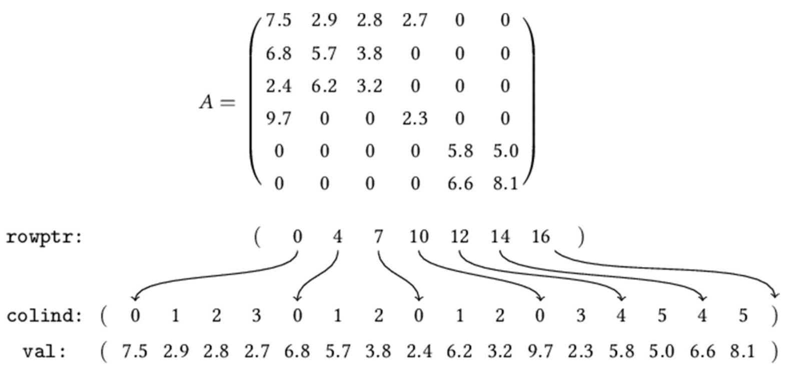
Normal and dense matrix are stored in computers as bi-dimensional arrays: each element $a_{i,j}$ is stored in the i-th row and j-th column. Storing an *n X m* matrix requires an amount of memory proportional to the number of its elements, so $n * m * (type\ of\ data\ stored)$.

Sparse matrix could be saved in memory using less bytes than the ones required for normal and dense matrixes: it is possible to completely represent a sparse matrix storing only the elements different from zero and some info related to them. By optimising memory consumption there is a trade-off: reconstructing from memory the original matrix and accessing an element becomes more complex.

There are different ways to store a sparse matrix, depending on the distribution of the non-zero entries and the operation required to be done on it, a representation can be preferred to another.

In the file given, *kmer_V4a.mtx*, the sparse matrix is stored using the *compressed sparse row* (*CSR*) format. In this representation a matrix is described using three one dimensional arrays that consist of:

- The value of the non-zero elements of the matrix, in the order of traversing the matrix row by row;
- The column indices of the non-zero elements;
- The row pointers, the index of the previous two arrays where a given row starts.


A =
7.5 2.9 2.8 2.7 0 0
6.8 5.7 3.8 0 0 0
2.4 6.2 3.2 0 0 0
9.7 0 0 2.3 0 0
0 0 0 0 5.8 5.0
0 0 0 0 6.6 8.1
rowptr: (0 4 7 10 12 14 16)
colind: (0 1 2 3 0 1 2 0 1 2 0 3 4 5 4 5)
val: (7.5 2.9 2.8 2.7 6.8 5.7 3.8 2.4 6.2 3.2 9.7 2.3 5.8 5.0 6.6 8.1)

This format permits fast row access and matrix-vector multiplication.

For an *n X m* matrix, the value and column indices arrays have a length equal to the number of non-zero elements rather than the row pointers array which has a length of *n*+1 elements, the last entry stores the number of non-zeros in the matrix.

To extract the i-th row of a sparse matrix represented through CSR we need to take each element of the arrays containing values and column indices between indices *row_pointers[i]* and *row_pointers[i+1]*.

For the *compressed sparse row* representation to use less memory than the normal representation when storing a *n X m* sparse matrix, the number of non-zero values should be less than: $NNZ < (n * (m-1) - 1)/2$.

Here's a little explanation:

The length of a *n X m* matrix is $n * m$ if represented normally and $2 * nnz + (n+1)$ if represented through CSR. $2 * nnz + (n+1) < n * m$ which follows $nnz < \frac{n*m-n-1}{2} = \frac{n*(m-1)-1}{2}$.

What is *Sparse Matrix-Vector Multiplication?*

A matrix-vector multiplication *M*X = Y* is a linear algebra binary operation that produces a vector *Y* from a matrix *M* and a vector *X*. To be correctly executed, the number of columns of the matrix *A* must be equal to the number of rows of the vector *X*. The resulting vector has a number of rows equal to the number of rows of the matrix *A*.

A matrix vector multiplication done with a dense *n X m* matrix is an operation that has a complexity of $O(n^3)$, by operating with sparse matrixes and using an appropriate matrix representation format, which supports matrix-vector multiplications, the complexity can be substantially reduced.

A dense *n X m* matrix multiplied by a *n* elements vector requires $n * (m-1)$ sums and $n * m$ multiplication.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

What is the *spmv-csr.c* program doing?

Before understanding how to perform a *Sparse Matrix-Vector Multiplication* in CUDA it is important to focus first on the simpler one-thread CPU implementation.

An example of such implementation can be found in the *spmv-csr.c* file, which contains a C one-thread SPVM implementation. The file contains various functions to implement the algorithm and other auxiliar functions, such as the *get_time*() function that determines the time since the Epoch and it is used to determine how many seconds the GPU executed the program.

The *read_matrix()* function reads the input file given and creates the three arrays describing efficiently the sparse matrix using the *compressed sparse row* format described above. In the program the column indices array is called *col_ind* (also *col_ind_t*), the values array is named *values* (also *values_t*) and the array with the pointers to the first element of each row is called *row_ptr* (also *row_ptr_t*).

In the main function, an input file with the matrix value (compliant with CSR representation) is required, then the *read_matrix()* function is called. An array *X* with a number of elements equal to the number of rows of the matrix is then created and filled with random values. Finally the CPU sparse matrix-vector multiplication function is called and the result is saved in another vector *Y* that has the same length of *X*.

The CPU implementation of the sparse matrix-vector multiplication performs a matrix-vector multiplication between the input matrix and the random generated array, saving the result in the *Y* array, but there is an important optimization: only the non-zero matrix rows elements are taken into consideration.

Thanks to the *row_ptr* array, that contains the index of the *value* array of the first non-zero element in the line, and the *col_ind* array, whose i-th element is the column index of the i-th element in the *value* array, the sum can be simplified in: values[j] * x[col_ind[j]], where *values[j]* are only the non-zero values of a row and *col_ind[j]* their column indices.

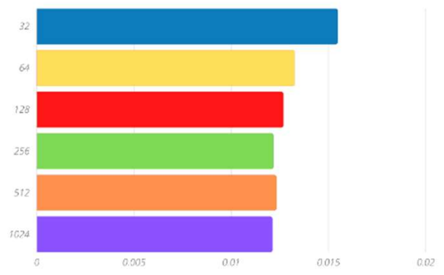
First implementation: one thread per matrix row

Since each element in the resulting array is independent by the others, a first way to parallelize the implementation in CUDA is to assign each row of the matrix to a different thread. Each thread is responsible for the multiplication of each non-zero element of a matrix row with the corresponding input vector elements.

An example of this naive CUDA implementation can be found in the ***simple-spmv.cu*** file. The algorithm implemented is similar to the one-thread CPU implementation of *spmv-csr.c*: since the three vectors describing the array need to be loaded, the *read_matrix()* function can be re-used, and also *get_time()* to measure the execution time of the GPU.

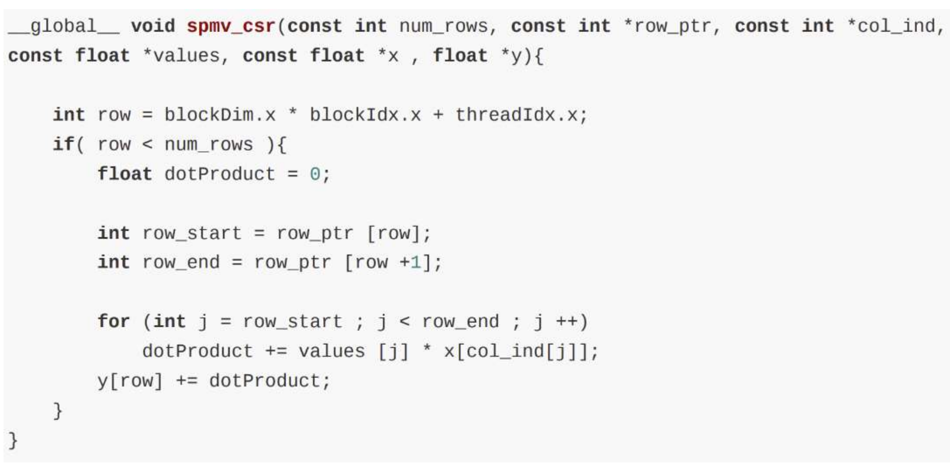
Of course, new variables and functions need to be used: in order to give GPU kernel variables, the *cudaMalloc()* and *cudaMemcpy(…, cudaMemcpyHostToDevice)* functions need to be invoked to allocate and copy data from CPU memory to GPU memory.

The *spmv_csr()* kernel launces a number of blocks equal to the number of matrix rows divided by the block dimension. In my environment (using Google Colab) the most efficient execution was with a block dimension of 1024, although there was a difference of less than 5% from the same test done with a block size of 256 and 512. Of course, the block size should always be a multiple of the warp size, 32.



Time (in seconds) required to perform matrix-vector multiplication using one thread per row. The graph presents the different execution times based on the number of threads used.

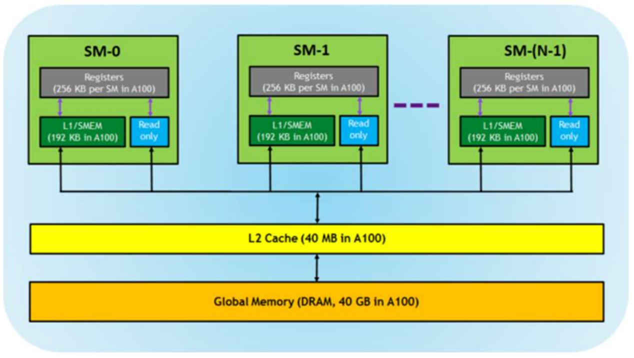
After the *spmv_csr()* kernel is called, a *cudaDeviceSynchronize()* needs to be done and finally the resulting data can be copied from the GPU memory to the CPU memory through a *cudaMemcpy(…, cudaMemcpyDeviceToHost)* and then deleted, *cudaFree()*.


```
__global__ void spmv_csr(const int num_rows, const int *row_ptr, const int *col_ind,
const float *values, const float *x , float *y){

    int row = blockDim.x * blockIdx.x + threadIdx.x;
    if( row < num_rows ){
        float dotProduct = 0;

        int row_start = row_ptr [row];
        int row_end = row_ptr [row +1];

        for (int j = row_start ; j < row_end ; j ++)
            dotProduct += values [j] * x[col_ind[j]];
        y[row] += dotProduct;
    }
}
```

A problem of this implementation, which reduces performance, can be the *warp divergence*: if the number of non-zero elements varies considerably between two near lines, different threads can take different times to complete the for loop execution. Some threads of the same warp run for more *for* iterations than others while others have to wait them in an inactive state.

Second implementation: one 32-thread warp per matrix row

The one thread per matrix row implementation has another big problem: if the input matrix has several elements per row, the implementation can cause a *high cache miss rate*.

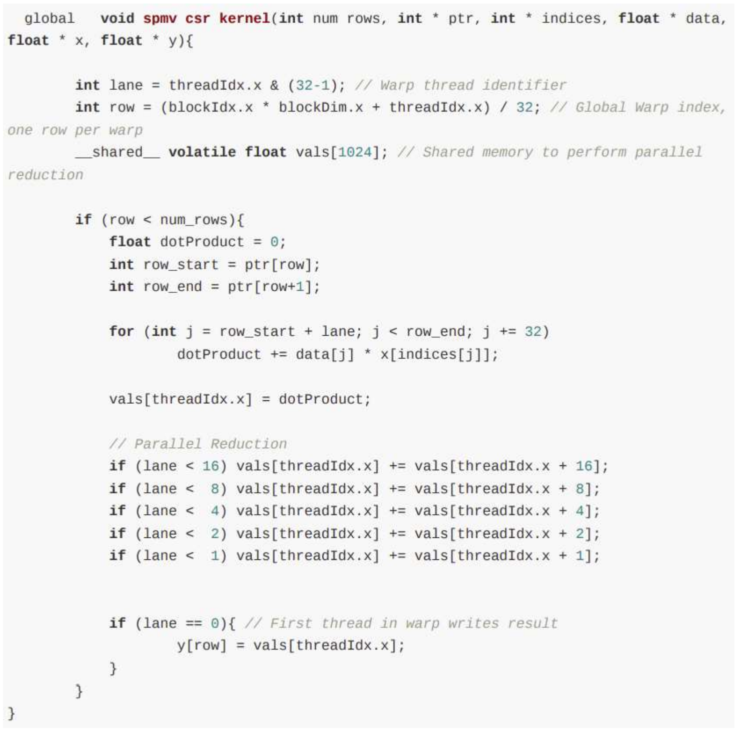


The GPU architecture has the following memory hierarchy: each thread has a private register; each streaming multiprocessor (the part of GPU responsible to execute each CUDA block) has a fast L1 cache that can be used as shared memory and a read only memory where constants and instructions lies; there is a L2 cache shared among all streaming multiprocessor and a global memory, the DRAM.

When a value is required by a kernel, it looks at every cache, from the nearest to the farthest, and in case the value is not found (*miss*), it needs to be loaded from the slow global memory and saved in all the caches that have produced a *miss*.

Accessing memories further away from threads and streaming multiprocessor increase the propagation and the execution times.

To reduce the number of cache lines used, and consequently the cache miss rate, multiple threads can be assigned to do the operation of a single matrix row. This can be achieved using a L1 shared memory: each thread can do the same operations as in the previous implementation and save the result in a shared memory. Thanks to shared memory the result can be saved through the more efficient parallel reduction, that instead of summing elements linearly $O(n)$ it sums them logarithmically $O(\log_2 n)$. The result is saved by only the first thread of each group. In order to work correctly the shared memory should have the same size of the block dimension.



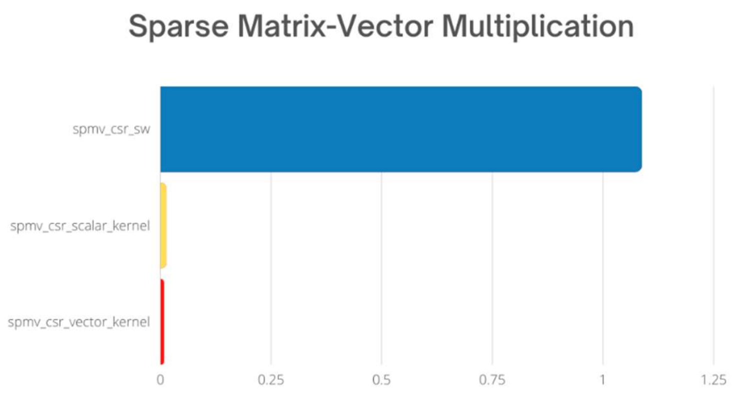
```
  global   void spmv csr kernel(int num rows, int * ptr, int * indices, float * data,
float * x, float * y){

        int lane = threadIdx.x & (32-1); // Warp thread identifier
        int row = (blockIdx.x * blockDim.x + threadIdx.x) / 32; // Global Warp index,
one row per warp
        __shared__ volatile float vals[1024]; // Shared memory to perform parallel
reduction

        if (row < num_rows){
            float dotProduct = 0;
            int row_start = ptr[row];
            int row_end = ptr[row+1];

            for (int j = row_start + lane; j < row_end; j += 32)
                    dotProduct += data[j] * x[indices[j]];

            vals[threadIdx.x] = dotProduct;

            // Parallel Reduction
            if (lane < 16) vals[threadIdx.x] += vals[threadIdx.x + 16];
            if (lane <  8) vals[threadIdx.x] += vals[threadIdx.x + 8];
            if (lane <  4) vals[threadIdx.x] += vals[threadIdx.x + 4];
            if (lane <  2) vals[threadIdx.x] += vals[threadIdx.x + 2];
            if (lane <  1) vals[threadIdx.x] += vals[threadIdx.x + 1];


            if (lane == 0){ // First thread in warp writes result
                    y[row] = vals[threadIdx.x];
            }
        }
}
```

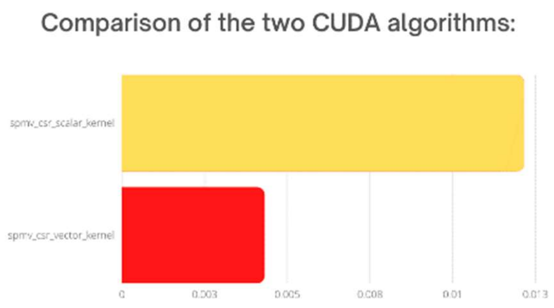
An implementation of the algorithm can be found in the ***spmv-csr.cu*** file, in which each block is divided into groups that have the same dimension of a warp (32) and each group of 32 threads operates on a single line. The correct number of blocks to be launched is $\frac{num\_rows}{block\ dimension/\ warp\ size}$,

Between thread operations, such as before or after parallel reduction, there's no need to synchronize threads with the *__syncthreads()* function: since we only execute inside a warp, there's no need to set others block level synchronization barrier that will only slow the execution.

In the first implementation the best-case scenario occurs when each element of the random generated vector is loaded from the slower global memory only once. The worst case scenario happens when the matrix is very sparse and only a few values are used for every cache line. In the second implementation the worst-case scenario doesn't cause problems: a group of thread use different cache lines whether they are reading values or writing the result. As a result, the cache miss rate is definitely lower.

The performance of the second CUDA implementation are up to three times better than the first GPU algorithm and up to 250 times better than the single thread CPU implementation. These results come from testing done using the Google Colab platform This is due to the more parallel operation done and the less cache miss rate.





Time taken to perform the matrix-vector multiplication, measured in seconds, in the three different cases: spmv-csr.c, simple_spmv.cu, spmv-csr.cu.