GPU computing homework
CSR group

Diego Oniarti - 257835
diego.oniarti@studenti.unitn.it

2024-2025

1 Introduction

The problem at hand is that of performing a *Sparse-Matrix Dense-Vector Multiplication* and implementing different optimizations through the use of the GPU.

The matrix will be read from a `.mtx` file and be converted into CSR format before being used for the multiplication. The time used for the conversion in CSR form will not be taken into consideration when measuring the time taken by the multiplications.

The repository with the code can be found at https://github.com/diego-oniarti/GPU_homework.

1.1 Abstractions

In the code the type of the elements in the matrix and the vector will be `data_t`, defined in a macro, and the matrix will be passed to the functions as a struct of this form:

```
typedef struct {
    data_t *vals;   // Vector containing the nonzero values
    int *xs;        // Vector containing the x indices of the nonzero elements
    int *ys;        // Vector containing the row pointer
    int nvals;      // Number of nonzero values
    int nrows;      // Number of rows
    int ncols;      // Number of columns
} MAT_CSR;
```

2 CPU algorithm

I started with a sequential implementation on the CPU as a baseline. The result of this implementation will be used to assess the correctness of the following ones.

> **Remark.** In this and other pseudocodes, trivial allocations and freeing of memory will be omitted for the sake of clarity and to center focus on the algorithm.

The values in the arrays `ys`, `xs`, `vals`, and `ret` are all accessed sequentially, making good use of caching. The slight exception to this is `ys`, where each element is read along with its successor (in the halt check of the inner loop). The values being accessed are however close together, so locality of reference is still taken advantage of.

The worst use of cache in this access pattern happens when accessing the vector, since values are read randomly from it.

Algorithm 1: CPU implementation

```
input  : MAT_CSR *mat, data_t *vec
output: data_t* result
//Iterate over the rows
for (int y=0; y<mat→ nrows; y++) do
    data_t row_acc = 0;
    //Iterate over the elements in the row
    for (int i=mat→ys[y]; i<mat→ys[y+1]; i++) do
        int x = mat→xs[i];
        row_acc += mat→vals[i] * vec[x];
    end
    result[y] = row_acc;
end
```

3 GPU algorithms

I implemented 4 different kernels to run on the GPU, each one improving in some aspect on the previous ones.

3.1 Thread Per Row

This first implementation mirrors the one done on the CPU, but instead of rows being accessed sequentially, a thread is generated to handle each of them individually. This implementation retains the locality of access

Algorithm 2: Thread Per Row

```
input  : data_t *ret, *vals, *vec
         int rows, *xs, *ys
int tid = blockIdx.x*blockDim.x + threadIdx.x; if (tid < rows) then
    //The elements in the row are summed linearly and sequentially
    data_t acc = 0;
    for (int i=ys[tid]; i<ys[tid+1]; i++) do
        acc += vals[i] * vec[xs[i]];
    end
    ret[tid] = acc;
end
```

advantages from its fully sequential counterpart, but its made much faster through parallelization.

3.2 Warp Per Row

The logical progression from the algorithm using one thread per row would be to use one thread per nonzero value. Using this solution it is quite easy to get the product of each element in the matrix with the corresponding one in the vector.
When summing the elements in a row, however, the synchronization is much harder. Having one thread sum up all the elements in the row would be inefficient, while doing a reduction in parallel requires the thread to synchronize possible across blocks.

My solution to this was to dedicate a whole warp of threads to each row. This way the synchronization is much easier, and the performance is still improved from the previous implementation since more threads are used.

To allow the sum of the elements in a row to be performed by multiple threads, the individual values are stored into a buffer. Parts of the buffer are then reduced and the result is put in the return array.

Algorithm 3: Warp Per Row

```
input  : data_t *vals, *vec, *ret, *buffer,
         int *xs, *ys, cols, rows
int tid = blockIdx.x*blockDim.x + threadIdx.x;
int wid = threadIdx.x / 32;
int friend_id = threadIdx.x % 32 ;                          //Thread's place in the warp
int row = blockIdx.x * (blockDim.x / 32) + wid;
buffer[tid] = 0 ;                          //Set to zero to avoid breaking the reduction
if (row < rows) then
    int start = ys[row];
    int end = ys[row+1];
    //Loop with stride equal to the warp size
    //Needed in case there are more elements in a row than threads in a warp
    data_t sum = 0;
    for (int i=start+friend_id; i<end; i+=32) do
        sum += vals[i] * vec[xs[i]];
    end
    buffer[tid] = sum;
end
//Perform the reduction over the warp
for (int s=1; s<32; s<<=1) do
    __syncthreads() ;                          //Wait for the previous iteration to be over
    if ((tid & ((s<<1)-1)) == 0) then
        buffer[tid] += buffer[tid+s];
    end
end
//The first thread in the warp is tasked with moving the result to the output
if (friend_id==0 && row<rows) then
    ret[row] = buffer[tid];
end
```

3.3 Adding shared memory

The final improvement I made to the algorithm was that of removing the buffer used for the reduction and perform that in shared memory instead.

4 Mode of testing

4.1 Output validity

As mention prior, the result of each execution is compared to that of the sequential CPU implementation. This had shown no errors for Algorithm 1. Errors in the result occur in the last two algorithms. I do however have good reason to attribute this error to the imprecision of floating point operations in C and to the fact addition is not commutative.
The proof I have is:

1. The maximum error decreases when switching `data_t` from `float` to `double`
2. The error completely vanishes if the matrix only contains integers
3. The error is consistent across runs and valgrind confirms no read from dirty memory occurs
4. The error is always very small, in the order of 10^{-16}

4.2 Timing

For the CPU implementation I've used the `gettimeofday` function to get the timing of the execution, while for the GPU kernels I made use of CUDA events.

Each algorithm has been tested with different block sizes, ranging from 32 to 1024 (the maximum allowed by the GPU).

Moreover, each of these runs has been repeated 13 times: 3 pre-runs and 10 actual runs. From the latter 10 runs I got the average time and the standard derivation in the values.

4.3 Datasets

The algorithms have been tested on 4 different matrices, while the vector has always been filled with ones. One of the matrices is generated on the spot, each cell having a 1% chance of being a random value from 1 to 10. The others have been taken from the UF sparse matrix collection.

name	rows	columns	nonzeros	(%)	type
lp_ganges[1]	1309	1706	6937	0.3106%	real
delaunay_n23[2]	8388608	8388608	50331568	7.152546e-5%	binary
Stanford_Berkeley[3]	683446	683446	7583376	0.001624%	binary
custom	300000	200000	-	1%	real

5 Results

[1] https://www.cise.ufl.edu/research/sparse/matrices/LPnetlib/lp_ganges
[2] https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/delaunay_n23
[3] https://www.cise.ufl.edu/research/sparse/matrices/Kamvar/Stanford_Berkeley