HW02 - Parallelization of Sinkhorn-Knopp Algorithm using GPU (CUDA)

Kaya Gokalp

May 4, 2021

1 INTRODUCTION

In this homework we were asked to parallelize Sinkhorn-Knopp algorithm which is used do construct D_C and D_R matrices such that for input matrix A,

$$D_C A D_R$$

is doubly scholastic. Doubly scholastic means sum of any column or row is equal to 1. The input, matrix A needs to be in some form to be able to find the answer but in this homework input matrix A is assumed to be satisfying those requirements. After implementing the algorithm correctly, parallelization techniques applied and results are discussed in following sections. Some sections (about csr and some part of the errorCheck) are taken from the first homework's report.

2 PREPROCESSING

SPARSE MATRIX REPRESENTATION USING CCS AND CRS: Since we are dealing with large matrices moving and operating on the whole 2-dimensional matrix array is too expensive, we are using a representation for our sparse matrices CCS and CRS. In this homework construction of the required arrays for these representations are done for us. So the pre-processing step is already done. Only initialization of d_r and d_c to 1 is added to preprocessing step.

3 ALGORITHM

3.1 General structure of the code

I used 32 blocks and 1024 thread per block in my CUDA code. To be able to synchronize everything and continue to use several blocks, I created 3 different kernels. First kernel is responsible for row sum operation. After that kernel ends, another kernel executes the col sum. After both the row sum and col sum finishes errorCheck kernel starts to work. In errorCheck kernel each

thread calculates a error value then they place their error values to shared memory. Then in the shared memory values are sorted out. I used a simple max reduction to achieve sorting. At the end each block's max value will be ready. So 32 max value needs to be sorted to find global maximum. I could achieve that with the same kernel but 32 is small enough to be sorted in the CPU. So I copied 32 values to host and sorted them in the host, finally the error is printed out.

3.2 General structure of the kernels

Total number of elements to be processed might get bigger than block * thread. So there might be some cases where each thread needs to process multiple elements. So I used a basic for loop in the kernels. Which starts from threads global id, increments by step size (which is 32*1024) while it is smaller than total number of elements to processed.

3.3 Using CCS and CRS representations in the algorithm

IN THE ALGORITHM WE ARE USING a_{ij} for calculating rsum (sum of row) and csum (sum of columm). Since we need to do multiplication between a_{ij} and $d_R[i]$ or $d_C[j]$ we need to somehow access a_{ij}. A simple trick is used for this purpose. Since all the elements in the matrices are either 0 or 1, the multiplication for all i or j simple becomes a summation of the $d_R[i]$ or $d_c[j]$ where the corresponding a_{ij} is non-zero. So basically to be able to calculate rsum or csum the position of the data in the currently iterating row or column is needed. This is obtained by looping from xadj[i] to xadj[i+1] while reaching the value of adj with corresponding loop index (for column same procedure but txadj and tadj). To demonstrate it more clearly this is the corresponding code block for row sum (where i goes from row 0 to row n):

```
int rowIndex = xadj[i];
for(;rowIndex<xadj[i+1]; rowIndex++)
{
  rsum += cv[adj[rowIndex]];
}
```

Corresponding code block for column sum (where i goes from column 0 to column n):

```
int colIndex = txadj[i];
for(;colIndex<txadj[i+1]; colIndex++)
{
  csum += rv[tadj[colIndex]];
}
```

Assuming the example matrix from Figure 3.1 is the input. An example run for the code block above goes like the following.

Row sum for row 0:

1. i = 0, rowIndex = xadj[0] which is 0
2. xadj[1] which is 3 so the loop will run from 0 to 3.
3. adj[0] = 0 which is the first non zero value's column in row 0, similarly adj[1] is 2 and adj[2] is 3. Similarly adj[1] and adj[2] is the second and third non zero value from row 0
4. Since a_{ij} is 1 when it is non-zero and from step3 non-zero valued indices are found in the row 0. So summation of d_j[adj[0]], of d_j[adj[1]] and of d_j[adj[2]] is our rsum. Which is 1+1+1 in the first run assuming d_j is all 1's.

Column sum for row 0:

1. i = 0, colIndex = txadj[0] which is 0
2. txadj[1] which is 2 so the loop will run from 0 to 2.
3. tadj[0] = 0 which is the first non zero value's row in col 0, similarly tadj[1] is 4.

4. Since a_{ij} is 1 when it is non-zero and from step3 non-zero valued indices are found in the col 0. So summation of d_i[tadj[0]] and d_i[tadj[1]] is the csum. Which is 1+1 in the first run assuming d_i is all 1's.

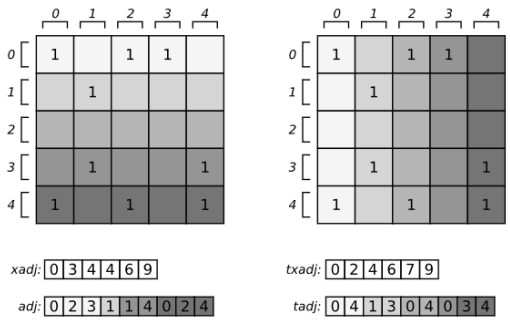

Figure 3.1: Example matrix with xadj, adj, txadj, tadj values

3.4 Calculating the error per iteration

WHILE CALCULATING THE ERROR for each row i, for any non-zero valued index a, d_C[a] is multiplied with d_R[i] and summed up. The resulting value should be 1 so 1 - the sum result is our error. The biggest error is selected as the error for the iteration. The following code block is used to calculate error for the iteration.

```
int step = 32*1024;
__shared__ double cache[1024];
int  index = blockDim.x * blockIdx.x + threadIdx.x;
for(int i = index; i<(*nov)-1; i+=step)
{
  double errorSum = 0;
    int row_start = xadj[i];
  for(int jj = row_start; jj<xadj[i+1]; jj++)
      errorSum += (cv[adj[jj]] * rv[i]);
        double errorVal = fabsf(1.0 - errorSum);
    cache[threadIdx.x] = cache[threadIdx.x] < errorVal ? errorVal :cache[
    threadIdx.x];
}
__syncthreads();
```

3.5 Max reduction

WITHIN THE SAME BLOCK WITH THE ERROR CALCULATION PART max error reduction is achieved with the following code block. The elements to be reduced are already placed into the shared memory. Max reduction part basically finds out the max element for each block. In the end there are 32 values that needs be sorted in order the find the global max. I could use another kernel with 32 threads and 1 block to reduce that to a single element. But I thought 32 elements is small enough to use CPU to find max. Since the element size is small I thought that would be faster.

```
int ib = blockDim.x / 2;
while(ib != 0)
{
  if(threadIdx.x < ib && sharedMem[threadIdx.x + ib] > sharedMem[threadIdx.
    x])
    {
      sharedMem[threadIdx.x] = sharedMem[threadIdx.x + ib];
```

```
    }
    __syncthreads();
    ib /=2;
}
  if(threadIdx.x == 0)
      errorOut[blockIdx.x] = sharedMem[0];
  sharedMem[threadIdx.x] = 0;
```

4 Run Times and Performance

4.1 NLPKKT240.MTXBIN

The table below shows the compared results with CPU and GPU executing with the same input. (nlpkkt240.mtxbin) 20 iteration. NVPROF output can be seen below.

1 thread	2 thread	4 thread	8 thread	16 thread	GPU
29.7125	20.9219	15.189	13.3875	16.35	7.23

Table 4.1: 20 iteration nlpkkt240

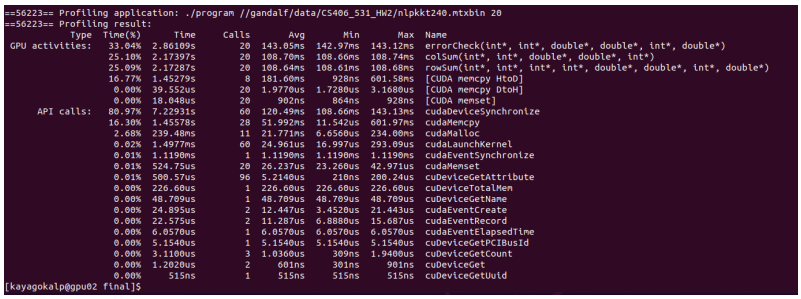

Figure 4.1: nvprof output for nlpkkt240

4.2 VAS_STOKES_1M.MTXBIN

The table below shows the compared results with CPU and GPU executing with the same input. (vas_stokes_1M.mtxbin) 20 iteration.

1 thread	2 thread	4 thread	8 thread	16 thread	GPU
2.78706	1.9483	1.4676	1.2410	1.3414	0.68

Table 4.2: 20 iteration vas_stokes_1M

4.3 VAS_STOKES_4M.MTXBIN

The table below shows the compared results with CPU and GPU executing with the same input. (vas_stokes_4M.mtxbin) 20 iteration.

1 thread	2 thread	4 thread	8 thread	16 thread	GPU
10.8042	7.5768	5.7593	4.9374	4.71605	2.7063

Table 4.3: 20 iteration vas_stokes_4M

4.4 STOKES.MTXBIN

THE TABLE BELOW shows the compared results with CPU and GPU executing with the same input. (nlpkkt240.mtxbin) 20 iteration.

1 thread	2 thread	4 thread	8 thread	16 thread	GPU
28.6803	20.1223	15.3656	15.2899	15.0598	7.27

Table 4.4: 20 iteration stokes

5 CONCLUSION

WHILE COMPLETING THIS HOMEWORK , I realized that utilizing a GPU is not an easy task to be done. Mainly because of the debugging process is harder than host code. But when the utilization is done the results are very good. I did not applied all the optimization I could apply to my code because of the timing constraints. But even this version is easily beats CPU based parallel code. I wanted to use async memory copy as a further improvement on the code. While working on the homework I heavily used nvprof, and I understood that it is a vital tool to have while working with CUDA code. I wanted to apply all the optimizations that I am aware of but I am also happy with the result I got which can be improved a lot more. One last thing that I found out while completing the homework is the block level synchronization is a issue to be aware of while writing CUDA code. I used 3 different kernels to achieve block level synchronization since kernels are executed sequentially.