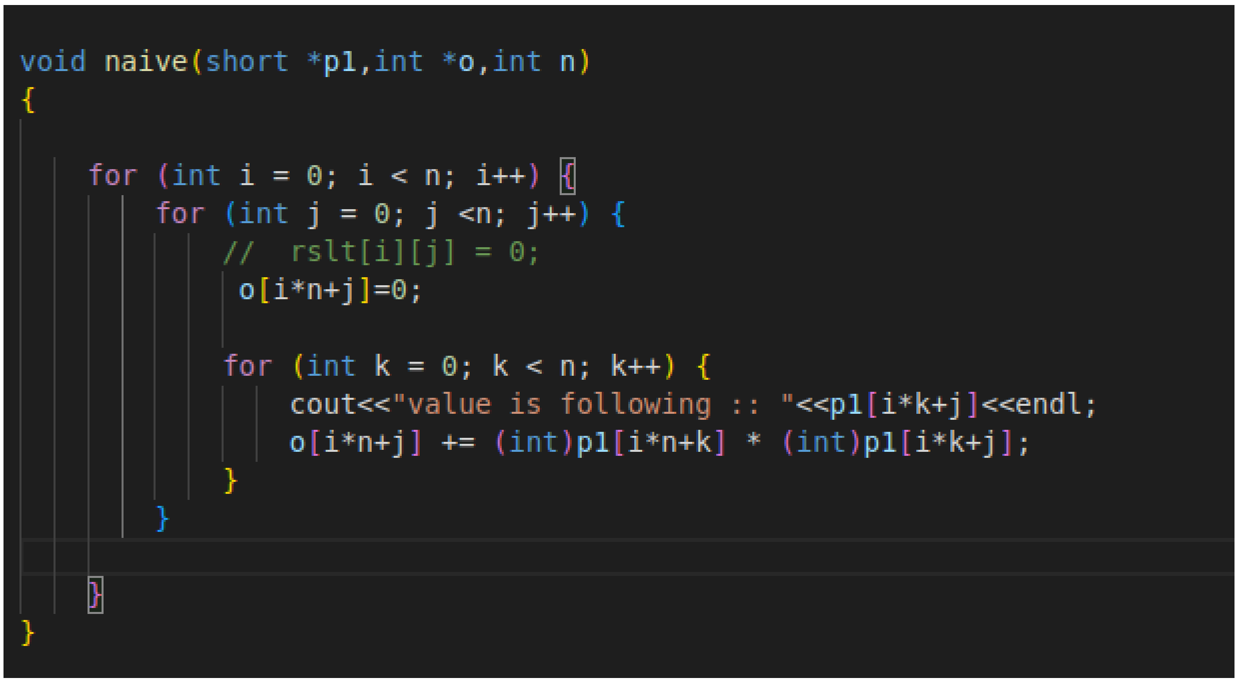
COL380
ASSIGNMENT 4 REPORT
ANKIT KUMAR(2020CS10323)

1. ALGORITHM USED:

(i) Naive approach::
So first of all i implemented the naive approach for the matrix multiplication without any threading and using the three nested loops so in that my algorithm is taking much time.and a small snippet of the naive approach used is pasted below:

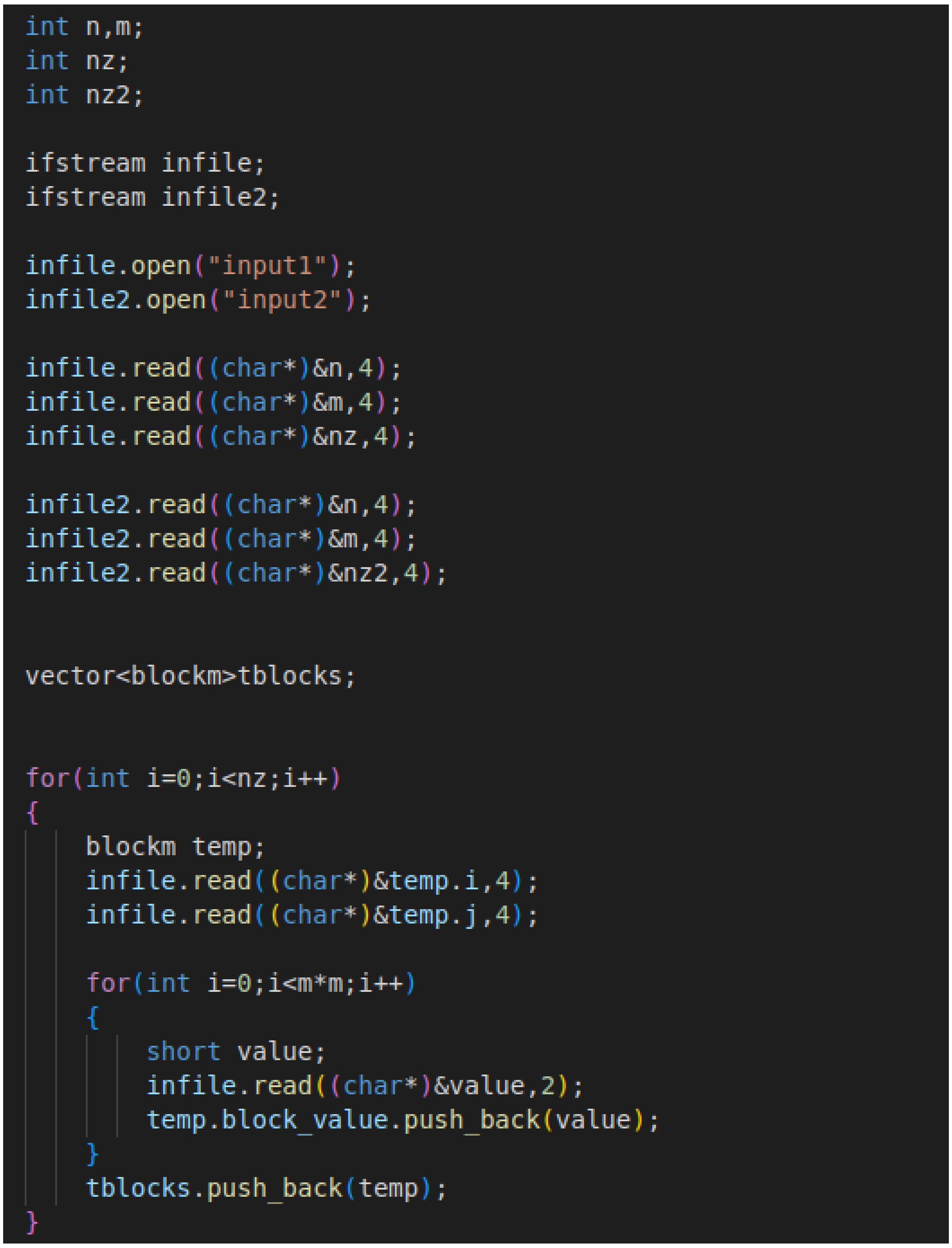


```
void naive(short *p1,int *o,int n)
{

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <n; j++) {
            //  rslt[i][j] = 0;
             o[i*n+j]=0;

            for (int k = 0; k < n; k++) {
                cout<<"value is following :: "<<p1[i*k+j]<<endl;
                o[i*n+j] += (int)p1[i*n+k] * (int)p1[i*k+j];
            }
        }


    }
}
```

Time complexity of the naive approach is O(n3) which is very bad as the size of n increases then our time also increases by huge amount.

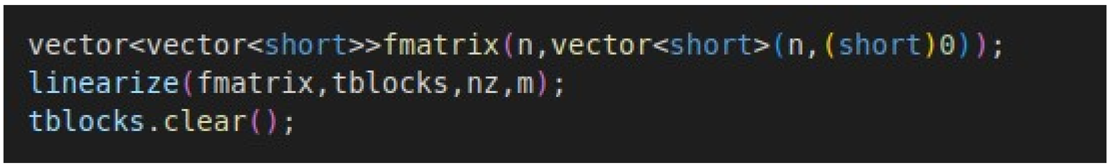
(ii) **Algorithm with the use of cuda environment:**

Basically in thi algorithm firstly i take the input from the file as described in the assignment store the data into block structure i made whose snippet is pasted below now we have all non zero blocks of the input matrices now i implemented a linearize function which convertes this block data into linear form so that i can pass the pointer to the cuda kernel function to compute upon. So the steps are given below:
1. Take the input in the form of blocks from the input files:

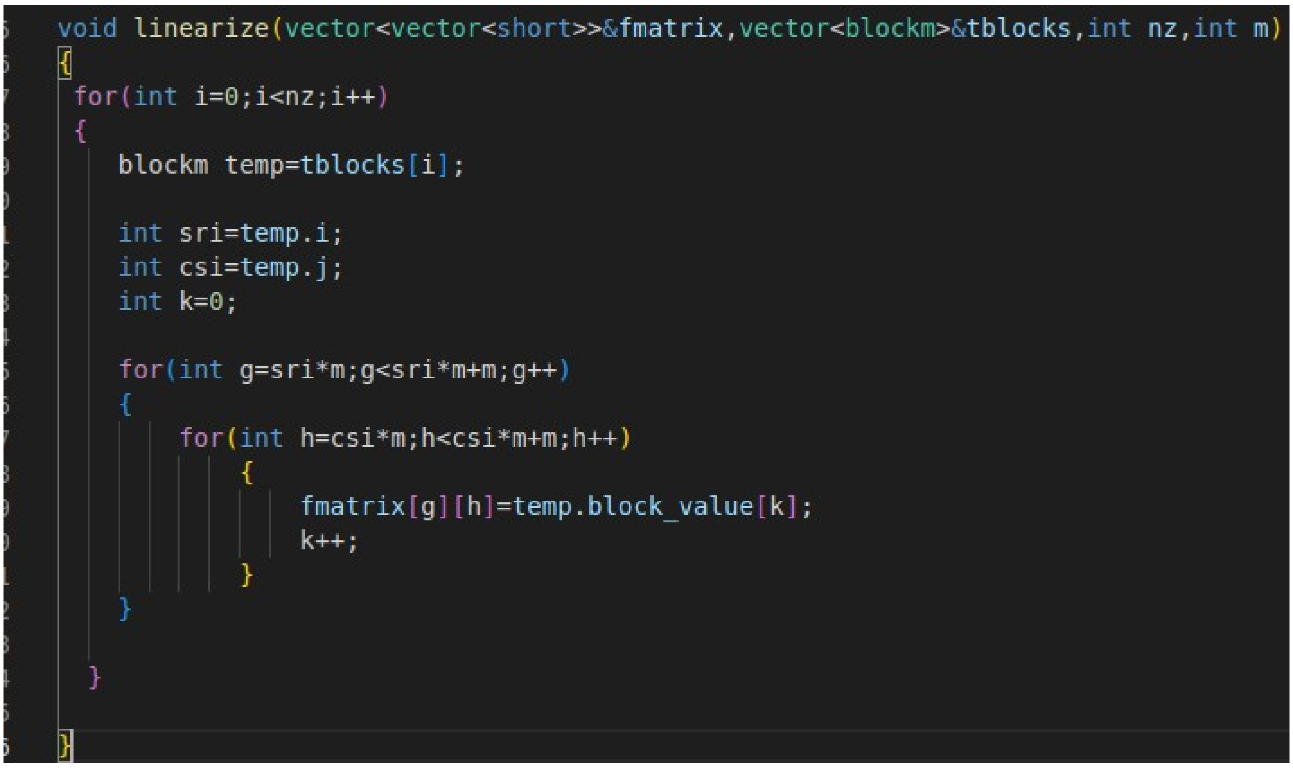


```
int n,m;
int nz;
int nz2;

ifstream infile;
ifstream infile2;

infile.open("input1");
infile2.open("input2");

infile.read((char*)&n,4);
infile.read((char*)&m,4);
infile.read((char*)&nz,4);

infile2.read((char*)&n,4);
infile2.read((char*)&m,4);
infile2.read((char*)&nz2,4);


vector<blockm>tblocks;


for(int i=0;i<nz;i++)
{
    blockm temp;
    infile.read((char*)&temp.i,4);
    infile.read((char*)&temp.j,4);

    for(int i=0;i<m*m;i++)
    {
        short value;
        infile.read((char*)&value,2);
        temp.block_value.push_back(value);
    }
    tblocks.push_back(temp);
}
```

2. After taking the inpus from the files i just linearized the blocks to make two one dimensional matrix fmatrix and fmatrix2 whose snippets is given below:

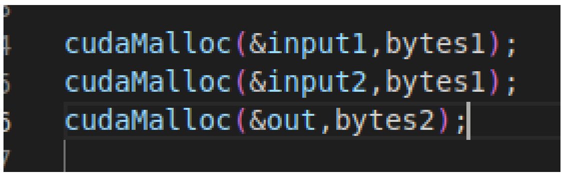


```
vector<vector<short>>fmatrix(n,vector<short>(n,(short)0));
linearize(fmatrix,tblocks,nz,m);
tblocks.clear();
```



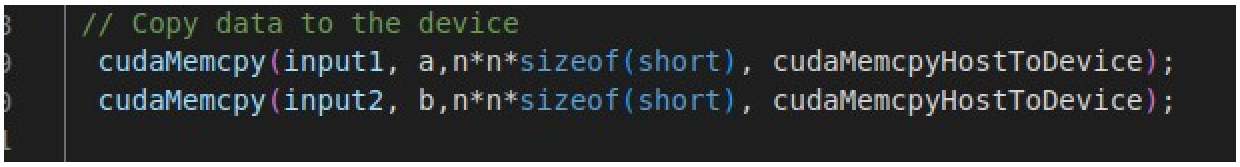
```
void linearize(vector<vector<short>>&fmatrix,vector<blockm>&tblocks,int nz,int m)
{
 for(int i=0;i<nz;i++)
 {
    blockm temp=tblocks[i];

    int sri=temp.i;
    int csi=temp.j;
    int k=0;

    for(int g=sri*m;g<sri*m+m;g++)
    {
        for(int h=csi*m;h<csi*m+m;h++)
            {
                fmatrix[g][h]=temp.block_value[k];
                k++;
            }
    }

  }

}
```

3.Now for initialising the cuda memory i used the cudamalloc function to initialise the two input matrices input1 and input2 of size n*n so that the matrixces can be fit into the gpu memory and also the datatype of the input matrices i kept as short but the datatype of the outptut matrix i kept as int.



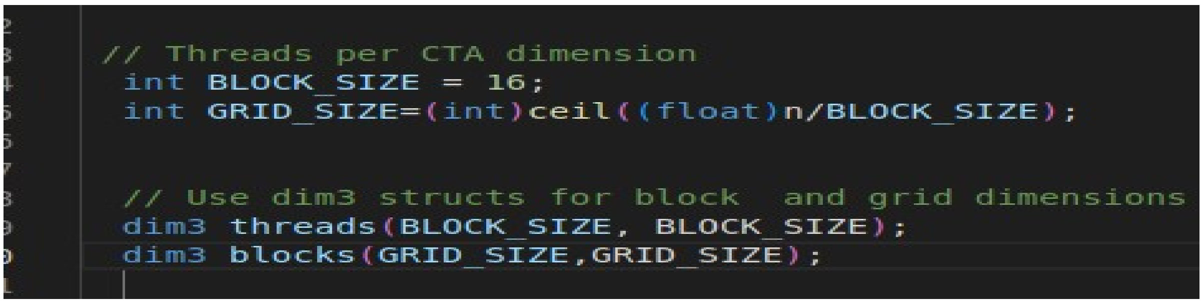
```
cudaMalloc(&input1,bytes1);
cudaMalloc(&input2,bytes1);
cudaMalloc(&out,bytes2);
```

4.After allocating the memory using cudamalloc i just copied down the data of input1 and input2 to the gpu's global memory by using cudamemcpy() function whose snippet is given below.



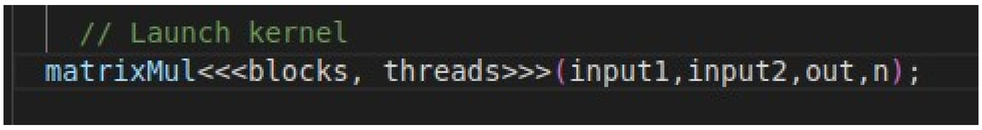
```
// Copy data to the device
 cudaMemcpy(input1, a,n*n*sizeof(short), cudaMemcpyHostToDevice);
 cudaMemcpy(input2, b,n*n*sizeof(short), cudaMemcpyHostToDevice);
```

5.Now to decide the number of threads per block used so i used 256 threads per block which is of dim3 so the threads are assigned as (16,16,1) and i took grid size equal to ceil(n/block size) i did this so that to evenly distribute n among the threads i created so the snippet of this is given below:

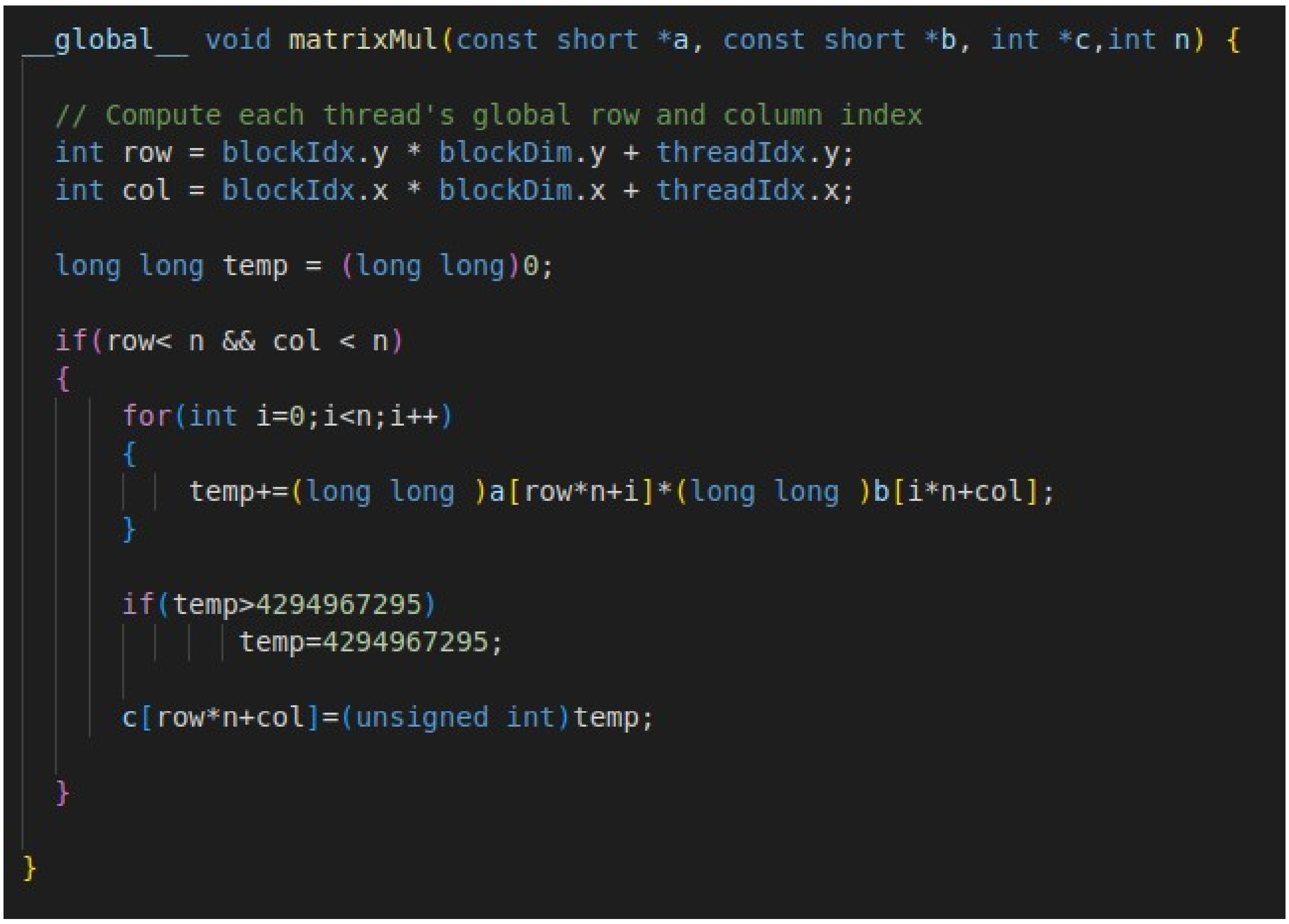


```
// Threads per CTA dimension
 int BLOCK_SIZE = 16;
 int GRID_SIZE=(int)ceil((float)n/BLOCK_SIZE);


 // Use dim3 structs for block  and grid dimensions
 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
 dim3 blocks(GRID_SIZE,GRID_SIZE);
```

6. Now i just call the kernel function in which i am   first calculating the row number by the formula : row=blockID.y*blockDim.y+threadidx.y; similarly for the column number and now afte this i just used a temporary variable long long temp whose initial value is kept as 0;
now after this i just used a while loop which first boundary check   then after just used the simple matrix multiplication loop to calculate the value at (i,j) of the output matrix the snippet of the above implementation is given below:

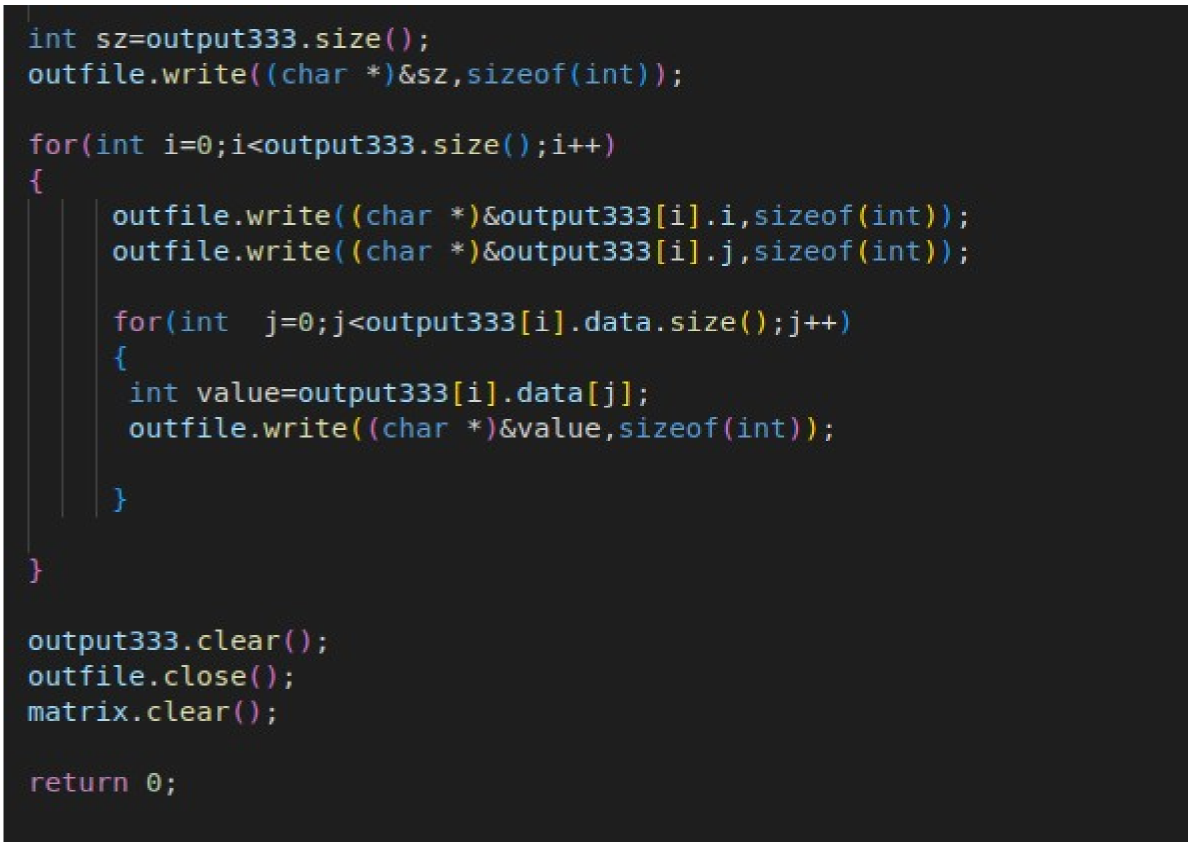


```
    // Launch kernel
  matrixMul<<<blocks, threads>>>(input1,input2,out,n);
```



```
__global__ void matrixMul(const short *a, const short *b, int *c,int n) {

  // Compute each thread's global row and column index
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  long long temp = (long long)0;

  if(row< n && col < n)
  {
      for(int i=0;i<n;i++)
      {
          temp+=(long long )a[row*n+i]*(long long )b[i*n+col];
      }

      if(temp>4294967295)
              temp=4294967295;

      c[row*n+col]=(unsigned int)temp;

  }

}
```

7.Now after calculating the result of the above matrix i just copied the resulted data from the gpu to cpu and then converted that linear resultant matrix into blocks and finally outputed to the desired file whose name is given in the command line of the main function.



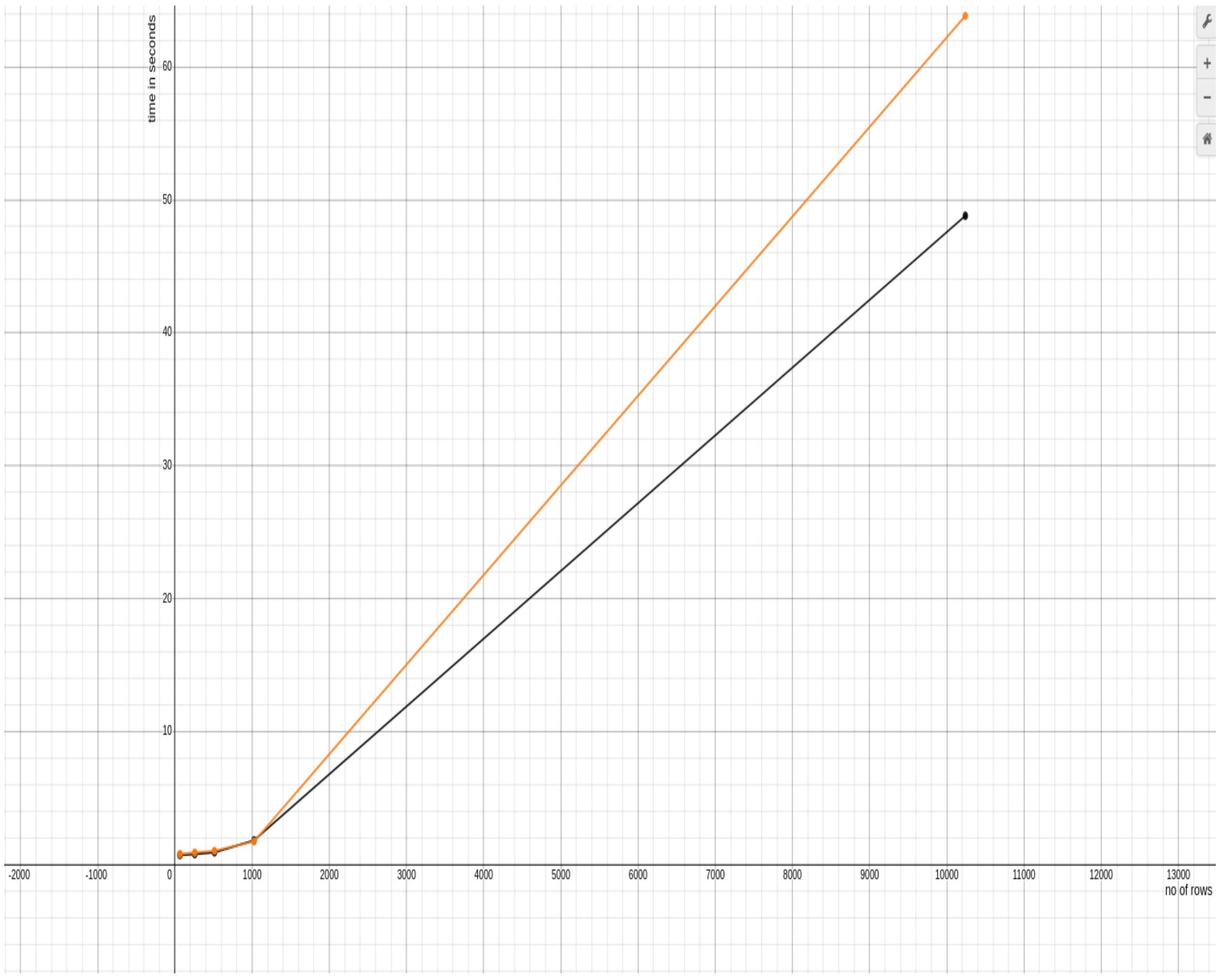
```
int sz=output333.size();
outfile.write((char *)&sz,sizeof(int));

for(int i=0;i<output333.size();i++)
{
    outfile.write((char *)&output333[i].i,sizeof(int));
    outfile.write((char *)&output333[i].j,sizeof(int));

    for(int  j=0;j<output333[i].data.size();j++)
    {
     int value=output333[i].data[j];
     outfile.write((char *)&value,sizeof(int));

    }

}

output333.clear();
outfile.close();
matrix.clear();

return 0;
```

Time complexity of the algorithm is still O(N3) but the work done is divided into many threads so throupghput is much higher below are some stats of the performance of my code::

2.PERFORMANCE NUMBERS OF THE ALGORITHM::

Number of rows in the matrix	Non zero blocks m=4	Non zero blocks m=8	m=4	m=8
64	256	64	0.730	0.808
256	4096	1024	0.804	0.918
512	16384	4096	0.938	1.049
1024	65536	16384	1.834	1.756
10240	655360	1638400	48.805	63.838

GRAPH OF PERFORMANCE:



here the lines in the black is for m=4 and the number of zero blocks as per the above table
here the lines in the green is for m=8 and the number of zero blocks as per the above table

x axis is the number of rows in the matrix y axis is the time taken.

THANK YOU