CS451 “Introduction to Parallel and Distributed Computing”

Homework 4

Jayanth Chidananda
A20517012

Compute capability and CUDA version

The GPU used for this code was **NVIDIA GeForce RTX 3050 T**i with the compute capability of 8.6, and the CUDA version used is **11.8**.
(**Note**: the program was designed on a laptop and not on the chameleon instance)

Code design

Initialization: The program takes four inputs the matrix size ‘N’ , the seed value for the random number generator for the matrix, with the seed value set the Matrix of size ‘N * N’ is initialized, the input matrix is A and the output matrix B is initialized to zero. It also takes the number of threads per block and number of blocks per grid to be used for computation.

Matrix: In this program the matrix is represented by a 1 D array, where a 2D matrix element of A[i][j] is present in the position A[i*size + j]. This was done so that moving data from GPU and host would be convenient.

```
void Initialize_Matrix()
{

  int row, col;
    A = (float *)malloc(sizeof(float)* N * N);
    B = (float *)malloc(sizeof(float)* N * N);
    test_B = (float *)malloc(sizeof(float)* N * N);
    for (row = 0; row < N; row++)
    {
        for (col = 0; col < N; col++) {

            A[row*N + col] = (float)rand() / 32768.0;
            B[row*N + col] = 0.0;
            test_B[row*N + col] = 0.0;
        }
    }
}
```

GPU allocation: The input matrix A present in the host , is copied to the GPU device. BY '**cudaMalloc**' function to allocate the space for the device matrix 'd_A' and then calling the '**cudaMemcpy**' to copy the data from host matrix A to device matrix 'd_A'. The same thing is done for output matrix B which is copied to device matrix 'd_B'.

```
printf("Copying input data from the host memory to the CUDA device. \n");
float *d_A = NULL;

//allocating space for device matrix
error = cudaMalloc((void **)&d_A, sizeof(float)*N*N);


if(error != cudaSuccess)
{
 fprintf(stderr,"Failed to copy vector A from host to device (error code %s)!\n",cudaGetErrorString(error));
 exit(EXIT_FAILURE);
}

//copying matrix data from host to device
error = cudaMemcpy(d_A,A,sizeof(float)*N*N,cudaMemcpyHostToDevice);

if(error != cudaSuccess)
{
 fprintf(stderr,"Failed to copy vector A from host to device (error code %s)!\n",cudaGetErrorString(error));
 exit(EXIT_FAILURE);
}
```

Threads and blocks: the number of threads per block is set, and the number of blocks per grid is set by default to 256 and 4 respectively, this can be varied by providing the values to the program through the command line.

```
void Initialize_parameters(int argc,char **argv)
{
  if(argc < 3)
  {
   printf("please enter seed and matrix size: program [matrix] [seed] \n");
   exit(0);
  }
  N = atoi(argv[1]);
  int seed = atoi(argv[2]);
  srand(seed);

  if(argc == 5)
  {
    blocksPerGrid = atoi(argv[3]);
    threadsPerBlock = atoi(argv[4]);
  }

}
```

Dividing Computation: the computation to be done by each thread is calculated by the below formula, which specifies the number of columns that each thread in the block needs to process. The 'divider' is set to a minimum value of 2 if the total number of threads is larger than total columns.

```
//calculating the no of columns each thread should normalize
int divider = (N/(threadsPerBlock*blocksPerGrid)) + 1;

// setting minimum columns to zero if total threads are more than columns
if(divider == 0)
{
 divider = 2;
}

printf("\n");
printf("performing matrix normalization with %d threads, and %d blocks. \n",threadsPerBlock,blocksPerGrid);
printf("No. of columns per thread is %d\n",divider);

//cuda kernel call

MatrixNorm<<<blocksPerGrid,threadsPerBlock>>>(d_A,d_B,N,divider);
error = cudaGetLastError();
```

Cuda Function: The cuda function is called by the above kernel call, each column of the matrix that is to be normalized is divided between the number of threads in each block. The portion of the columns that a specific thread in a specific block is supposed to normalize is calculated by the formula,

```
int startCol = (blockDim.x * blockIdx.x + threadIdx.x)*divider;
int endCol = (startCol + divider) > N ? N : (startCol + divider);
```

The blockDim.x gives the dimension of the block, since the grid is 1 Dimensional, this provides the number of blocks, 'blockIdx.x' gives the index of the block, and 'threadIdx.x' provides the index of the thread in that block.

Normalization: After the column values are known, each thread in all the blocks normalizes the specific portion of columns, and sets the values in the device matrix 'd_B'.

```
//cuda kernel function that runs on GPU cores
__global__ void MatrixNorm(float *matA,float *matB,int N,int divider)
{
    int row, col;
    float mu, sigma;
    //calculating the portion of rows each thread should normalize

    int startCol = (blockDim.x * blockIdx.x + threadIdx.x)*divider;
    int endCol = (startCol + divider) > N ? N : (startCol + divider);

    //performing normalization
    if(startCol< N)
    {
    for(col=startCol; col < endCol;col++)
    {
     mu = 0.0;
        for (row=0; row < N; row++)
            mu += matA[row*N + col];
        mu /= (float) N;
        sigma = 0.0;
        for (row=0; row < N; row++)
            sigma += powf(matA[row*N + col] - mu, 2.0);
        sigma /= (float) N;
        sigma = sqrt(sigma);
        for (row=0; row < N; row++) {
            if (sigma == 0.0)
                matB[row*N + col] = 0.0;
            else
                matB[row*N + col] = (matA[row*N + col] - mu) / sigma;
        }
    }
    }

}
```

Output: after the kernel function finishes, the device matrix ‘d_B’ is copied to the output matrix ‘B’, with cudaMemcpy function, and B is the normalized matrix, which is our result.

Verifying Correctness

The Cuda program was verified to be correct by comparing the output of the normalized matrix values of the serial function provided with that of the cuda function, with the same matrix size and seed value.

The serial function is run serially on the host device and not on the GPU, this ensures to find any error in parallel GPU computation.

The following function verifies the output of the cuda with that of the serial function, the output of the serial function is stored in matrix test_B.

```
void verify()
{
  printf("\n");
  printf("verifying \n");
  Normalize_serial();
int row, col;
 for (row = 0; row < N ; row++)
   {
     for(col = 0; col < N ; col++)
     {
        if(fabs(B[row*N + col] - test_B[row*N + col]) > 1e-5 )
        {
          printf("%f and %f in %d and %d\n",B[row*N + col],test_B[row*N + col],row,col);
                correct = false;
                return;
        }
     }
   }
}
```

Scaling

The CUDA program scales effectively, the following is the elapsed time for different thread and block numbers for matrices of different sizes.

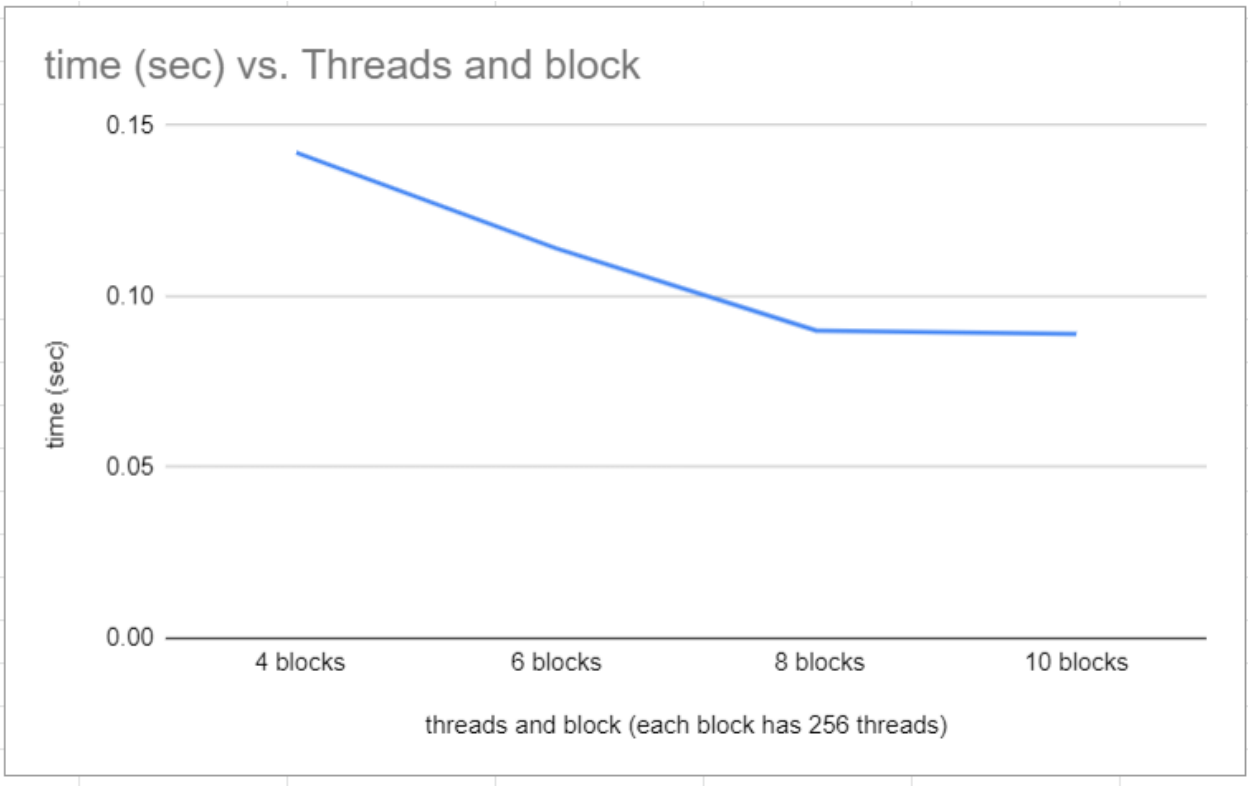
Varying number of blocks per grid

For a matrix of size, 8000. Setting **the number of threads in each block to a constant of 256**, and increasing the number of blocks per grid, we can observe that the

program's execution time reduces dramatically until a certain point, after which it only reduces slightly.

Threads and block	time (sec)
4 blocks	0.142
6 blocks	0.114
8 blocks	0.09
10 blocks	0.089

The table above shows the execution time of the CUDA program as block number is varied.

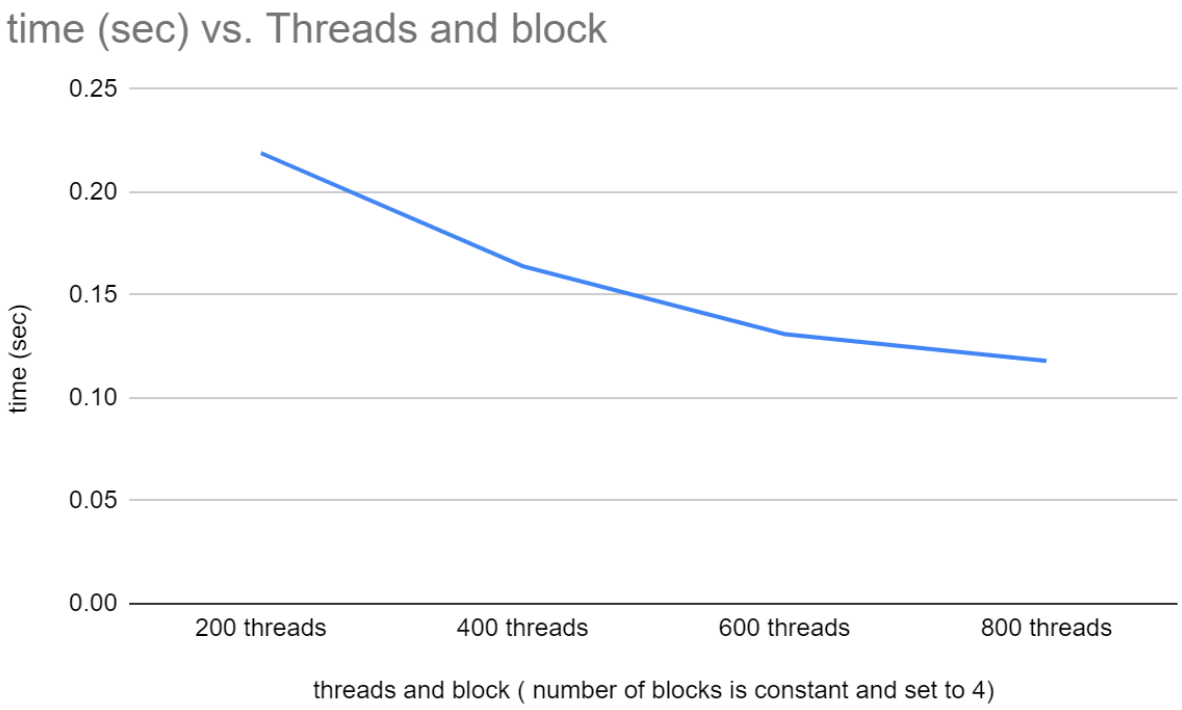


Varying number of threads per block

For a matrix of size 10000, keeping the **number of blocks to a constant of 4**. We can vary the number of threads in a block to observe effective scaling. The program's execution time reduces considerably when the number of threads per block increases.

The table below shows the execution time of the CUDA program with respect to the number of threads per block.

Threads and block	**time (sec)**
200 threads	0.219
400 threads	0.164
600 threads	0.131
800 threads	0.118

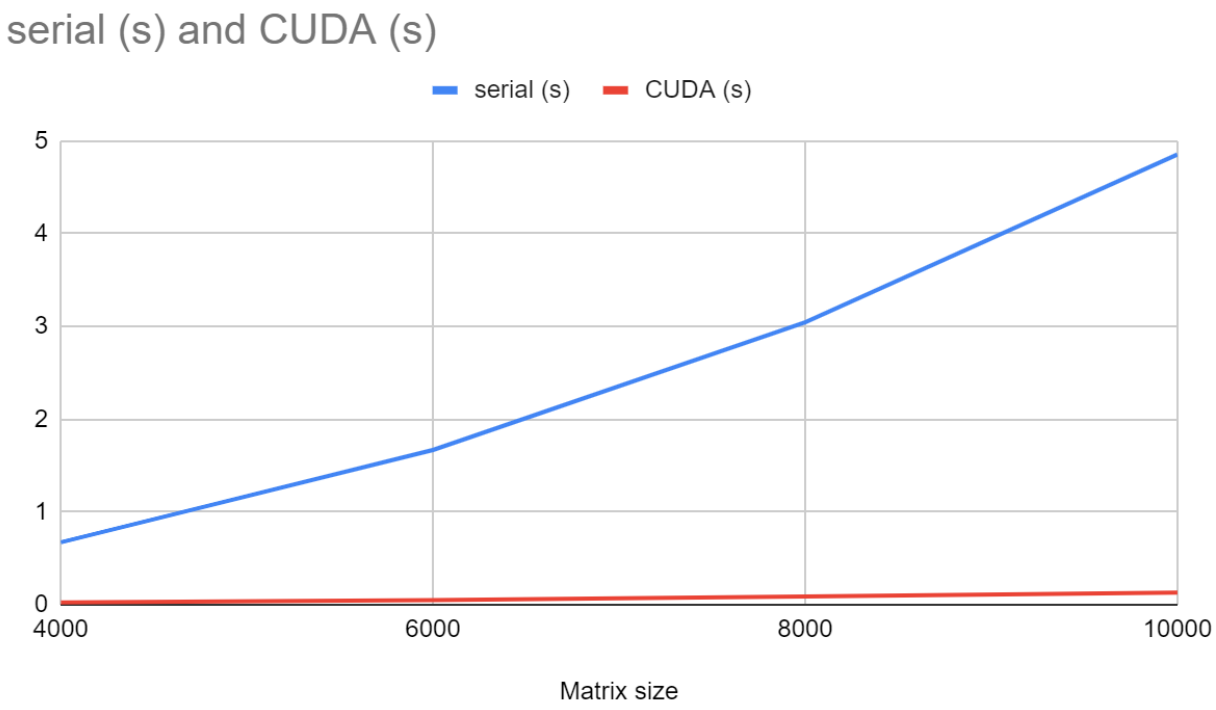


By observing the following output data, we can conclude that our program effectively scales as the number of resources available increases.

Code Efficiency

To determine whether the CUDA program is efficient, we can compare the program execution of the CUDA program to perform normalization against that of the serial program, for different matrix sizes.
Following is the graph, which shows the execution time for both the programs.
The number of threads per block for the CUDA program is set to 256 threads and 10 blocks.

As we can observe, the execution time of the CUDA program is significantly shorter than the serial program, so we can conclude that our CUDA program is efficient.



Matrix size	serial (s)	CUDA (s)
4000	0.672	0.023
6000	1.668	0.05
8000	3.043	0.088
10000	4.854	0.131

Above table, denotes the execution time taken by each program for different matrix sizes.

Output

The Following screenshots show some of the output of the program for verification.

```
C:\Personal Folders\MS Abroad\IIT Chicago\first sem\parallel processing\fourth assignment\submission>matcuda 10000 0 5 600
Copying input data from the host memory to the CUDA device.

performing matrix normalization with 600 threads, and 5 blocks.
No. of columns per thread is 4
Copying output data from the CUDA device to the host memory.

verifying
output is correct
output matrix


done.
Runtime = 0.131000 s.
```

```
C:\Personal Folders\MS Abroad\IIT Chicago\first sem\parallel processing\fourth assignment\submission>matcuda 10000 0 5 800
Copying input data from the host memory to the CUDA device.

performing matrix normalization with 800 threads, and 5 blocks.
No. of columns per thread is 3
Copying output data from the CUDA device to the host memory.

verifying
output is correct
output matrix


done.
Runtime = 0.118000 s.
```

```
C:\Personal Folders\MS Abroad\IIT Chicago\first sem\parallel processing\fourth assignment\submission>matcuda 10000 0 5 200
Copying input data from the host memory to the CUDA device.

performing matrix normalization with 200 threads, and 5 blocks.
No. of columns per thread is 11
Copying output data from the CUDA device to the host memory.

verifying
output is correct
output matrix


done.
Runtime = 0.219000 s.
```