<u>COL 380 Assignment 4</u>

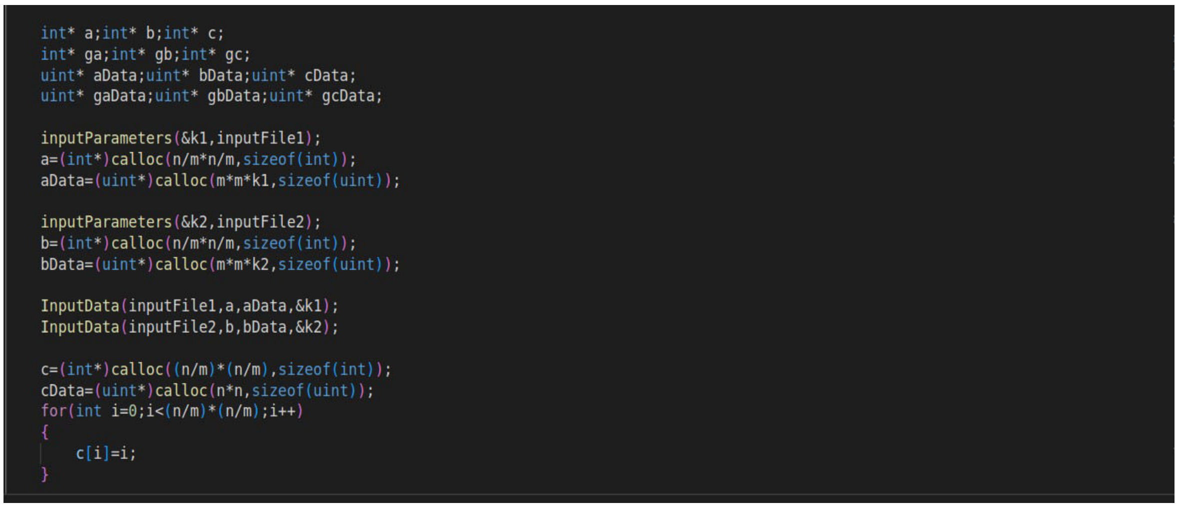
Name-Sibasish Rout
Entry no-2020CS10386

Table of Contents

Design and Implementation

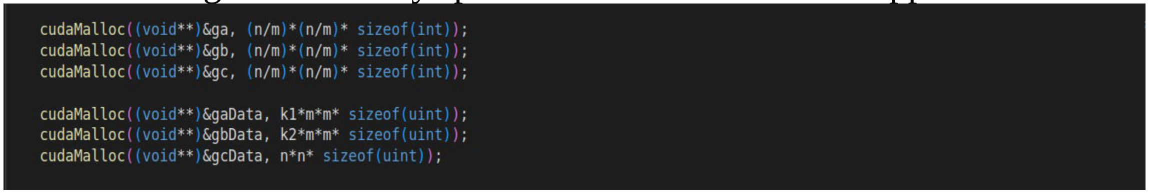
The multiplication basically invoves the use of pointers to store the matrix data and then multiplication using the concept of blocks. The steps involved are:-

1. First we take input using fstream from the binary files and feed them into thwo arrays.The second array contains the actual data of the matrix blocks and the first array contain a pointer to the position of the corresponding block in the data array.These arrays are stored as 1D arrays and so whenever we need to find the position of a particular particular block in the matrix or of a particular element we need to calculate the position using the size of each block and the total number of blocks possible. This would involve complex pointer arithimetic.



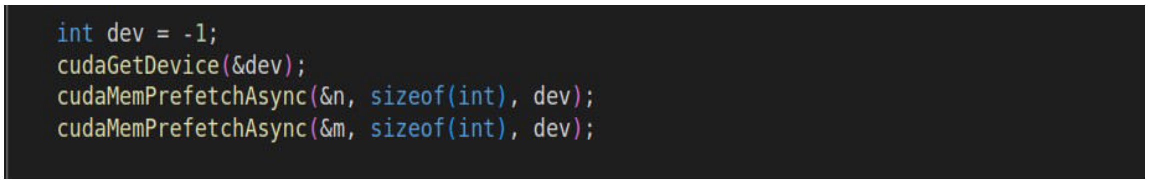
```
int* a;int* b;int* c;
int* ga;int* gb;int* gc;
uint* aData;uint* bData;uint* cData;
uint* gaData;uint* gbData;uint* gcData;

inputParameters(&k1,inputFile1);
a=(int*)calloc(n/m*n/m,sizeof(int));
aData=(uint*)calloc(m*m*k1,sizeof(uint));

inputParameters(&k2,inputFile2);
b=(int*)calloc(n/m*n/m,sizeof(int));
bData=(uint*)calloc(m*m*k2,sizeof(uint));

InputData(inputFile1,a,aData,&k1);
InputData(inputFile2,b,bData,&k2);

c=(int*)calloc((n/m)*(n/m),sizeof(int));
cData=(uint*)calloc(n*n,sizeof(uint));
for(int i=0;i<(n/m)*(n/m);i++)
{
    c[i]=i;
}
```

2. The second step involves declaring pointer to space in the GPU global memory using the cudaMalloc function. This is used to declare pointers for both the position arrays and actual data arrays for both the input as well as the output matrix in the global memory space of the device.All this happens in the host.



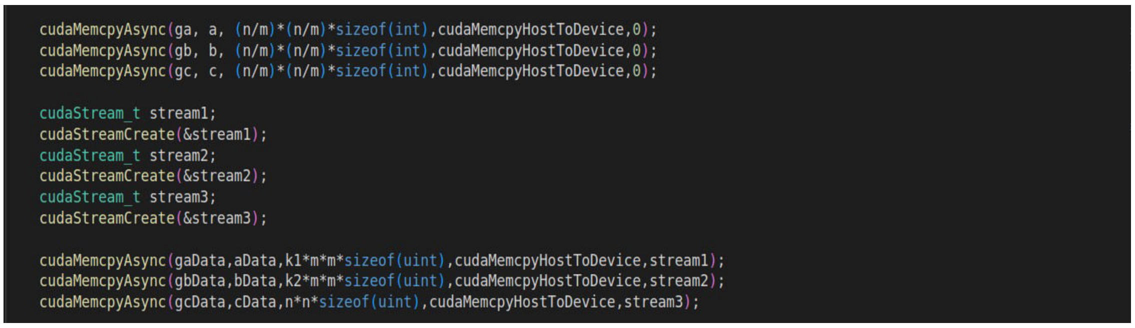
```
cudaMalloc((void**)&ga, (n/m)*(n/m)* sizeof(int));
cudaMalloc((void**)&gb, (n/m)*(n/m)* sizeof(int));
cudaMalloc((void**)&gc, (n/m)*(n/m)* sizeof(int));

cudaMalloc((void**)&gaData, k1*m*m* sizeof(uint));
cudaMalloc((void**)&gbData, k2*m*m* sizeof(uint));
cudaMalloc((void**)&gcData, n*n* sizeof(uint));
```

3. The values of n and m are declared as __managed__ and they are thus prefetched as they are involved in important computation in the future.



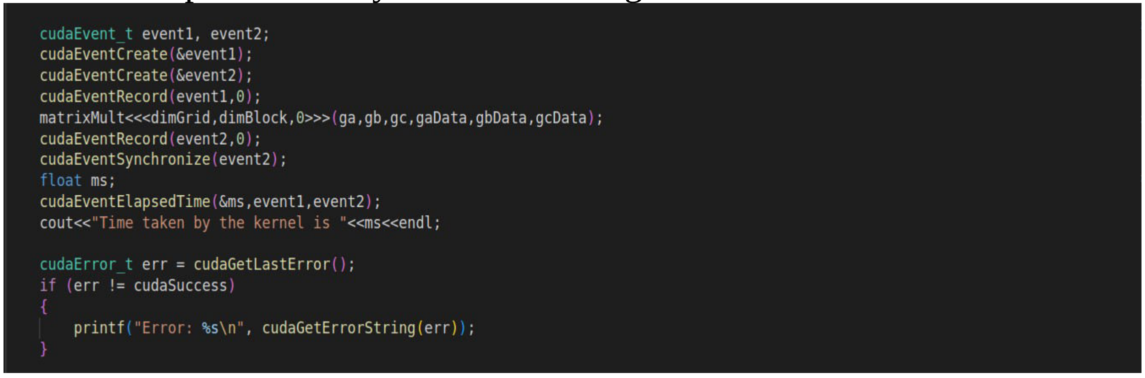
```
int dev = -1;
cudaGetDevice(&dev);
cudaMemPrefetchAsync(&n, sizeof(int), dev);
cudaMemPrefetchAsync(&m, sizeof(int), dev);
```

4. The values of position array and data array are then passed to the space previously allocated using cudaMalloc through dedicated streams to the device global memory so that they can be utilised by the kernel.The use of asynchronous copy and streams can be useful in reducing bottleneck due to copying the data.



```
cudaMemcpyAsync(ga, a, (n/m)*(n/m)*sizeof(int),cudaMemcpyHostToDevice,0);
cudaMemcpyAsync(gb, b, (n/m)*(n/m)*sizeof(int),cudaMemcpyHostToDevice,0);
cudaMemcpyAsync(gc, c, (n/m)*(n/m)*sizeof(int),cudaMemcpyHostToDevice,0);

cudaStream_t stream1;
cudaStreamCreate(&stream1);
cudaStream_t stream2;
cudaStreamCreate(&stream2);
cudaStream_t stream3;
cudaStreamCreate(&stream3);

cudaMemcpyAsync(gaData,aData,k1*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream1);
cudaMemcpyAsync(gbData,bData,k2*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream2);
cudaMemcpyAsync(gcData,cData,n*n*sizeof(uint),cudaMemcpyHostToDevice,stream3);
```

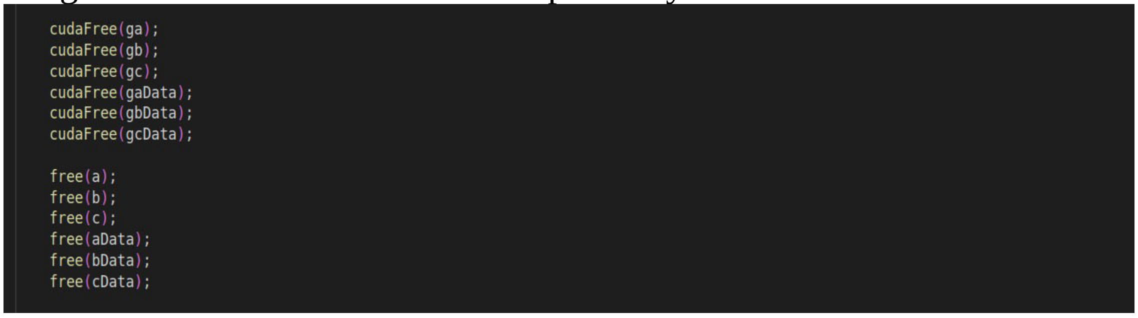
5. Finally the kernel function is called which is the most important part of the entire code. The kernel function determines the threadId and then assigns it work.Here as the value of m can be only 4 or 8 so we can easily give the work of multiplying blocks to find the result to one thread. Thus each block of the output matrix is given to a specific thread. This thread goes through the rows of blocks of the first matrix and columns of blocks in the second matrix(provided their existance is proved by the position array) and multiplies them to get the final result for the block. If the block is nonzero at the end its corresponding value in the position array is set to non negetive value otherwise it is set to -1.



```
cudaEvent_t event1, event2;
cudaEventCreate(&event1);
cudaEventCreate(&event2);
cudaEventRecord(event1,0);
matrixMult<<<dimGrid,dimBlock,0>>>(ga,gb,gc,gaData,gbData,gcData);
cudaEventRecord(event2,0);
cudaEventSynchronize(event2);
float ms;
cudaEventElapsedTime(&ms,event1,event2);
cout<<"Time taken by the kernel is "<<ms<<endl;

cudaError_t err = cudaGetLastError();
if (err != cudaSuccess)
{
    printf("Error: %s\n", cudaGetErrorString(err));
}
```

6. Here we use the help of a supporter function which given two blocks can compute theri result and store the answer at desired place in the resultant matrix.

7. In this step we copy the results back to the host from the device by using a cudamemcopyAsync() function again. This is then writtenv to the output file in the format as specified in the problem statement.Finally the allocated memories on the device global memory as well as the host RAM are freed by using cudafree and free functions respectively.

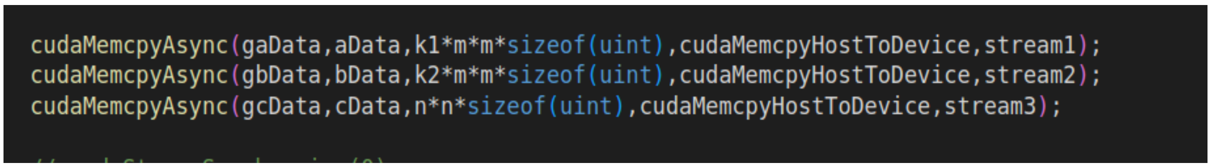


```
cudaFree(ga);
cudaFree(gb);
cudaFree(gc);
cudaFree(gaData);
cudaFree(gbData);
cudaFree(gcData);

free(a);
free(b);
free(c);
free(aData);
free(bData);
free(cData);
```

Optimisations

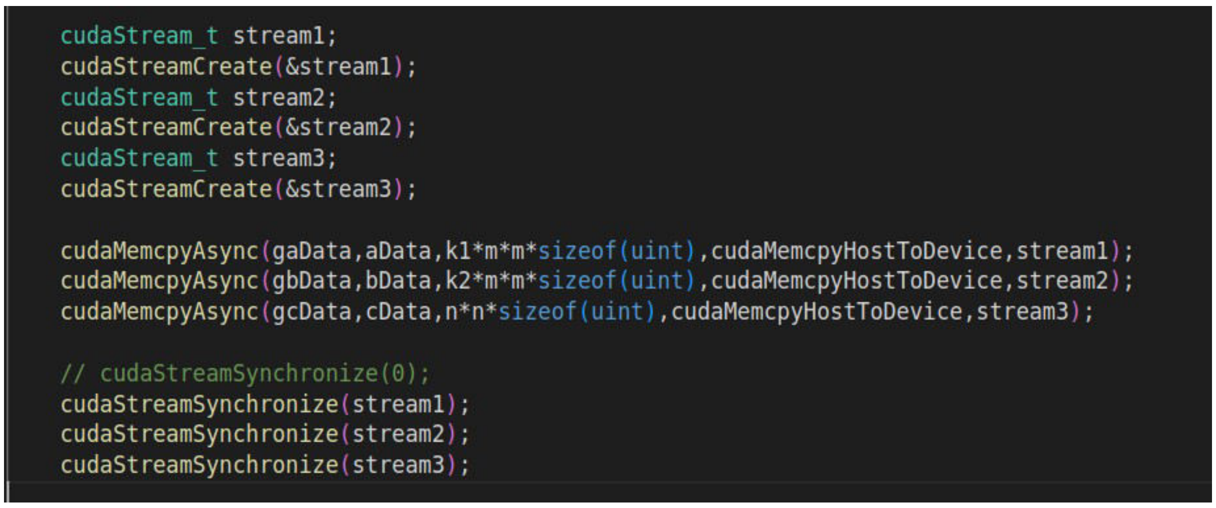
The optimisations done are

1. Use of cudaMemcpyAsync() instead of cudaMemcpy increases the performance in cases where we were doing large data transfers of the matrix data array as we can perform additional computations simultaneously with the memory copy operations.This will reduce the overhead significantly and would also allow us to reduce the bottleneck.



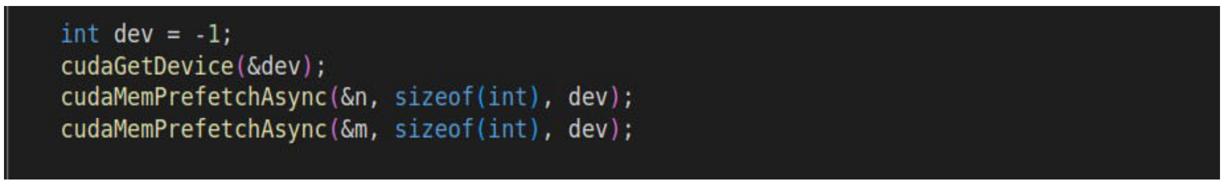
```
cudaMemcpyAsync(gaData,aData,k1*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream1);
cudaMemcpyAsync(gbData,bData,k2*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream2);
cudaMemcpyAsync(gcData,cData,n*n*sizeof(uint),cudaMemcpyHostToDevice,stream3);
```

2. Use of streams provide us with a huge huge advantage as now data transfer can happen in parallel and can reduce the communication bottleneck significantly.Thus the use of streams to transfer the position array and the data array values can be quite helpful as the value of n around 32768 can result in pretty large matrix data files and thus to tackle the communication involving these files strams can be used.



```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
cudaStream_t stream2;
cudaStreamCreate(&stream2);
cudaStream_t stream3;
cudaStreamCreate(&stream3);

cudaMemcpyAsync(gaData,aData,k1*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream1);
cudaMemcpyAsync(gbData,bData,k2*m*m*sizeof(uint),cudaMemcpyHostToDevice,stream2);
cudaMemcpyAsync(gcData,cData,n*n*sizeof(uint),cudaMemcpyHostToDevice,stream3);

// cudaStreamSynchronize(0);
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);
cudaStreamSynchronize(stream3);
```

3. Use of prefetching is useful in case of managed variables which will be accessed in future.There would be little overhead to fetch these variables during computation and hence prefetching their values can be quite useful.



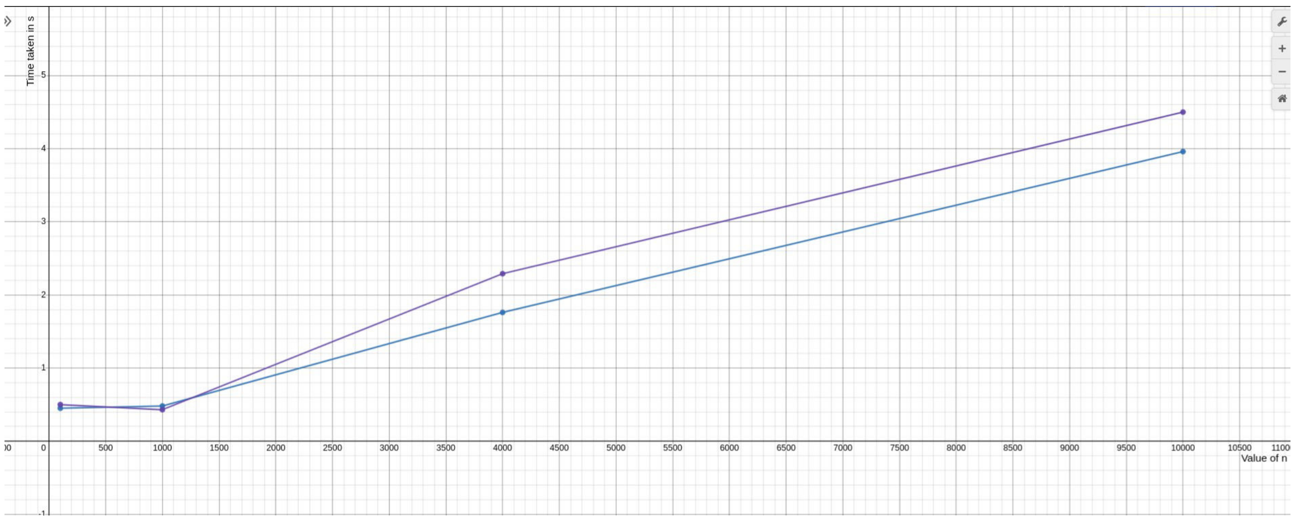
```
int dev = -1;
cudaGetDevice(&dev);
cudaMemPrefetchAsync(&n, sizeof(int), dev);
cudaMemPrefetchAsync(&m, sizeof(int), dev);
```

Analysis

Observations

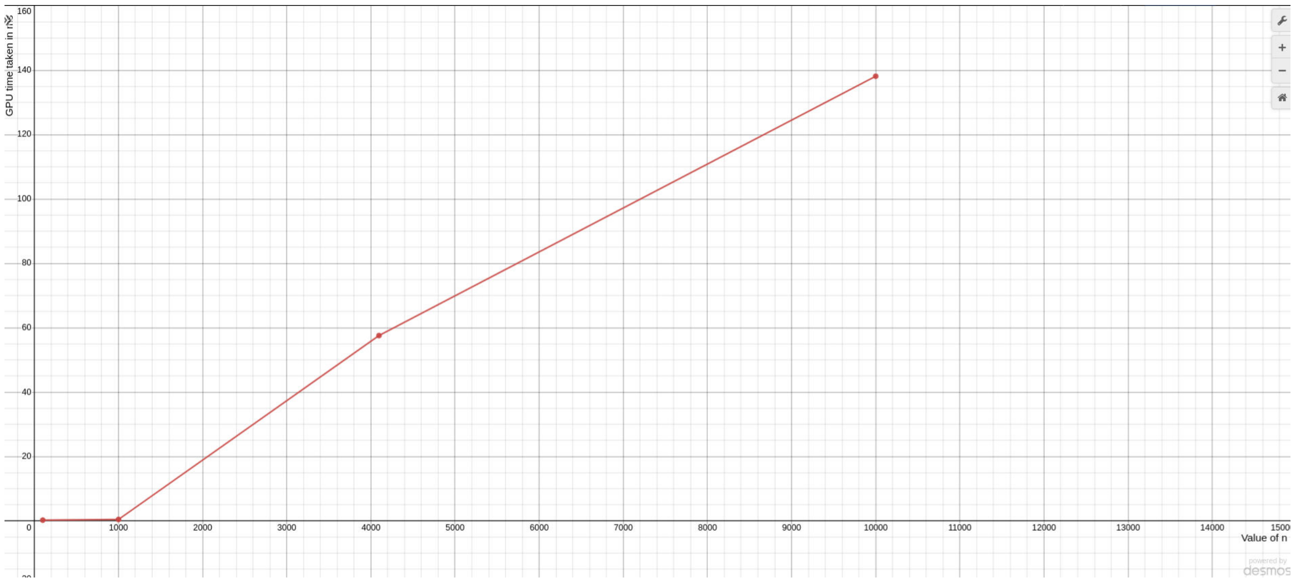
Value of n	Value of m	Value of k1	Value of k2	File size of input1	File size of input2	Output k	Total time	GPU time
8	4	4	4	172B	172B	4	0.45s	0.251ms
100	2	40	60	652B	972B	51	0.45s	0.265ms
1000	2	400	600	6.4KB	9.6KB	487	0.48s	1.632ms
10000	2	4000	6000	64KB	96KB	4804	3.96s	1001.42ms
4096	8	52450	52450	7.1MB	7.1MB	262144	1.76s	57.582ms
32768	4	65300	65300	2.3MB	2.3MB	518873	22.73s	4347.2ms
32768	8	16800	16800	2.6MB	2.6MB	68517	11.92s	574.749ms
100	4	40	60	1.6KB	2.4KB	87	0.50s	0.268ms
1000	4	400	600	16KB	24KB	930	0.43s	0.46048ms
10000	4	4000	6000	160KB	240KB	9598	2.29s	138.145ms

Plot of time taken vs value of n



blue line is for m=4, purple line m=8
From the graph we can see the time increases exponentially with the value of n and also for greater values of m the time is usually larger.

Plot for GPU time vs value of n(m=4)



Thus the value of GPU time also increases exponentially with the value of n. Hence the GPU takes exponentially more time to perform the same relative sized task as the value of n is varied linearly.