ASSIGNMENT PART-B REPORT

Mukul Sharma(Sr. no. 17935)

Aim : Cuda Version of the optimized Part A Diagonal Matrix Multiplication(DMM) program.

System Configuration : Intel(R) Xeon(R) Gold 6142 CPU@2.60 GHz (Server GPU).

Language: Cuda Programming

Cuda Functions Used:

1) **cudaMalloc(&d_x, N*sizeof(float)):** Its functionality is the same as malloc function of C program, used to allocate memory in device area and return the pointer of that location.
2) **cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice):** Using this method, we can copy from the device array pointed to by d_x to the host array pointed to by x by using cudaMemcpy with cudaMemcpyDeviceToHost.
3) **cudaFree(d_y):** Used to clean up the allocated memory.

Commands Used:

```
nvprof ./diag_mult_server data/input_4096.in
```

nvprof command tells the time spent in all GPU activities. Using this command, we get to know about the exact time spent by GPU in executing the program other than copying the arrays from CPU memory to GPU memory.

Optimisation Idea:

I have assigned one thread to each element of the output array.Each thread will execute independently and parallely. Hence, the execution time of the program will decrease drastically.

We can divide the given array into blocks and each block into some fixed number of threads. We can divide the arrays into blocks and threads in 1D, 2D or 3D fashion. Each thread of a particular block executes parallely.

I splitted the given output array of length 2*N-1 in 1D fashion. Let's assume that the number of threads per block is ‘M’. Then, the number of blocks in the output array will (2*N-1 + (M-1))/M. Therefore,

threadIdx.x = M and blockIdx.x = (2*N-1 +(M-1)) /M

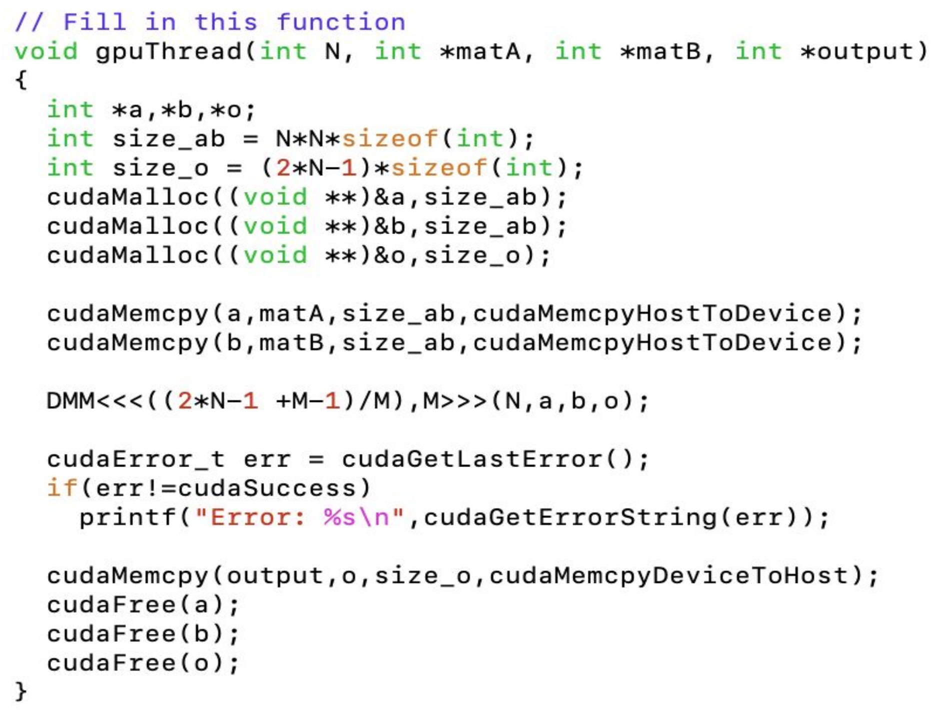
where M is in the order of power of 2.

I took M = 16 in my cuda implementation of the optimised DMM program.
As we know that there are two different kinds of memory **host** and **device** memory. To operate on GPU we first have to copy all array elements from Host to Device using the **cudaMemcpy** method of the cuda language.
Following **Code snippet 1** performing mainly three tasks:

1) Copy the required array elements from the Host to Device memory.
2) All independent threads calling the **kernel DMM**
3) After all threads execute the kernel, then the required output array will be copied from Device to Host memory.

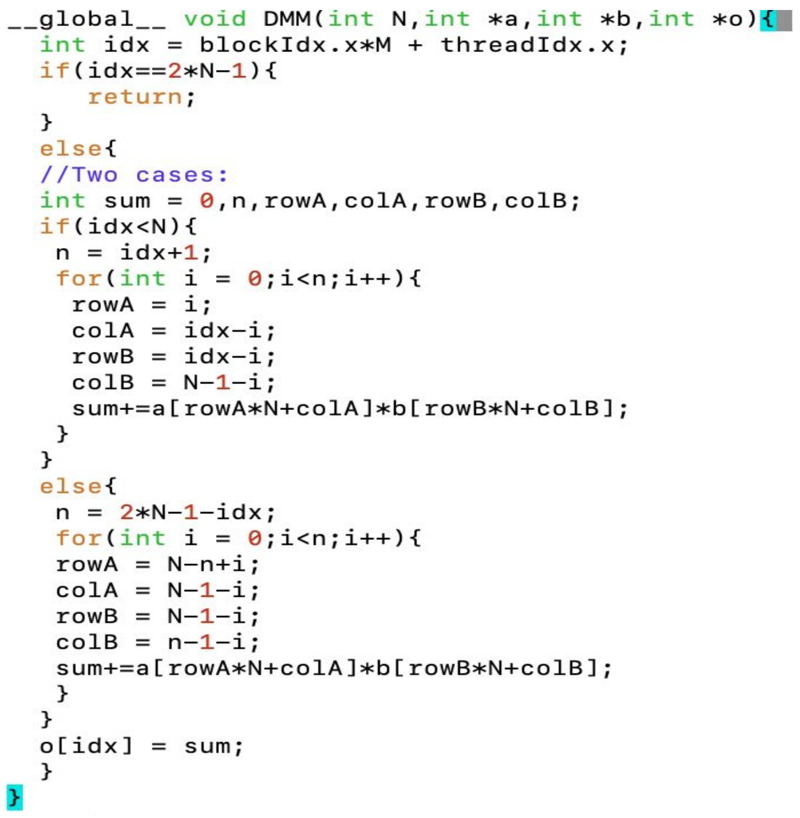
Code snippet 2 is showing the kernel DMM implementation. Using the **threadIdx.x** and **blockIdx.x**, we can first calculate the index of the output array. One thread is assigned to every element of the output array. After calculating the index, we can diagonally multiply elements of matrix matA and matB and store it in the output array.In this different cores of the GPU are used.

Code Snippet 1



```
// Fill in this function
void gpuThread(int N, int *matA, int *matB, int *output)
{
  int *a,*b,*o;
  int size_ab = N*N*sizeof(int);
  int size_o = (2*N-1)*sizeof(int);
  cudaMalloc((void **)&a,size_ab);
  cudaMalloc((void **)&b,size_ab);
  cudaMalloc((void **)&o,size_o);

  cudaMemcpy(a,matA,size_ab,cudaMemcpyHostToDevice);
  cudaMemcpy(b,matB,size_ab,cudaMemcpyHostToDevice);

  DMM<<<((2*N-1 +M-1)/M),M>>>(N,a,b,o);

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

  cudaMemcpy(output,o,size_o,cudaMemcpyDeviceToHost);
  cudaFree(a);
  cudaFree(b);
  cudaFree(o);
}
```

Code Snippet 2 (Kernel Code)



```
__global__ void DMM(int N,int *a,int *b,int *o){
  int idx = blockIdx.x*M + threadIdx.x;
  if(idx==2*N-1){
     return;
  }
  else{
  //Two cases:
  int sum = 0,n,rowA,colA,rowB,colB;
  if(idx<N){
   n = idx+1;
   for(int i = 0;i<n;i++){
    rowA = i;
    colA = idx-i;
    rowB = idx-i;
    colB = N-1-i;
    sum+=a[rowA*N+colA]*b[rowB*N+colB];
   }
  }
  else{
   n = 2*N-1-idx;
   for(int i = 0;i<n;i++){
   rowA = N-n+i;
   colA = N-1-i;
   rowB = N-1-i;
   colB = n-1-i;
   sum+=a[rowA*N+colA]*b[rowB*N+colB];
   }
  }
  o[idx] = sum;
  }
}
```

Observations:

Execution time table:

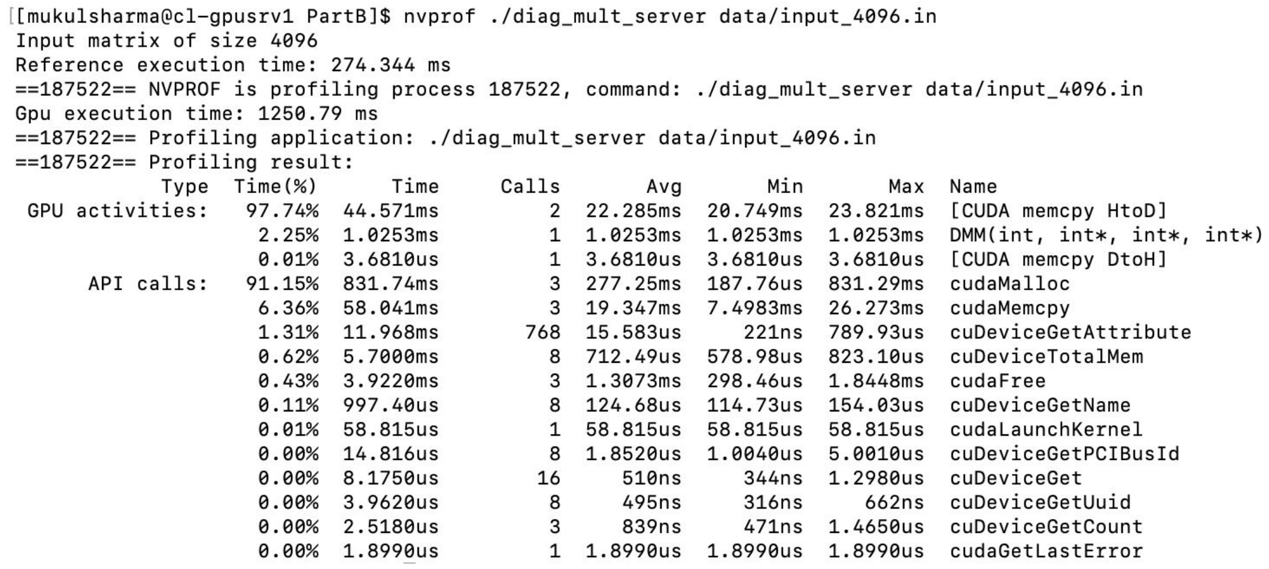
Input Size	**Reference time**	**GPU Exec. time**	**Speed Up**
n = 4096	274 ms	1250 ms	0.219
n = 8192	1613 ms	1306 ms	1.235
n = 16384	7723 ms	1651 ms	4.677

Why does the execution time in GPU increase from CPU, when input size is 4096?
Because GPU execution time includes all the GPU activities time like memory copy from host to device, kernel execution time etc. As we can see in the following **screenshot** 97% of the total execution time spent in copying data and only 2.25% of the total execution time spent in

executing kernel for all the threads. This shows that how the real execution time decreases drastically from the reference time 274ms to 1.025 ms.

We can also see from the above table pattern. As we increase the input size, the speed up also increases exponentially.

Command : nvprof ./diag_mult_server data/input_4096.in



```
[[mukulsharma@cl-gpusrv1 PartB]$ nvprof ./diag_mult_server data/input_4096.in
 Input matrix of size 4096
 Reference execution time: 274.344 ms
 ==187522== NVPROF is profiling process 187522, command: ./diag_mult_server data/input_4096.in
 Gpu execution time: 1250.79 ms
 ==187522== Profiling application: ./diag_mult_server data/input_4096.in
 ==187522== Profiling result:
             Type  Time(%)      Time     Calls       Avg       Min       Max  Name
  GPU activities:   97.74%  44.571ms         2  22.285ms  20.749ms  23.821ms  [CUDA memcpy HtoD]
                     2.25%  1.0253ms         1  1.0253ms  1.0253ms  1.0253ms  DMM(int, int*, int*, int*)
                     0.01%  3.6810us         1  3.6810us  3.6810us  3.6810us  [CUDA memcpy DtoH]
       API calls:   91.15%  831.74ms         3  277.25ms  187.76us  831.29ms  cudaMalloc
                     6.36%  58.041ms         3  19.347ms  7.4983ms  26.273ms  cudaMemcpy
                     1.31%  11.968ms       768  15.583us     221ns  789.93us  cuDeviceGetAttribute
                     0.62%  5.7000ms         8  712.49us  578.98us  823.10us  cuDeviceTotalMem
                     0.43%  3.9220ms         3  1.3073ms  298.46us  1.8448ms  cudaFree
                     0.11%  997.40us         8  124.68us  114.73us  154.03us  cuDeviceGetName
                     0.01%  58.815us         1  58.815us  58.815us  58.815us  cudaLaunchKernel
                     0.00%  14.816us         8  1.8520us  1.0040us  5.0010us  cuDeviceGetPCIBusId
                     0.00%  8.1750us        16     510ns     344ns  1.2980us  cuDeviceGet
                     0.00%  3.9620us         8     495ns     316ns     662ns  cuDeviceGetUuid
                     0.00%  2.5180us         3     839ns     471ns  1.4650us  cuDeviceGetCount
                     0.00%  1.8990us         1  1.8990us  1.8990us  1.8990us  cudaGetLastError
```

Conclusion: We can see the advantage of using GPU over CPU, as the number of cores increase the execution time of the multithread program decreases.