Lab Assignment 2

20232241 Dookyung Kang

- ✓ For entire answers, let M = numARows, N = numAColumns = numBRows, K = numBColumns. numAColumns must be equal to numBRows for matrix multiplication, it makes sense to perform multiplication of A (M * N) and B (N * K) to obtain C (M * K).
- ✓ Answer 1~3 refers to my version of code which is descripted on answer 5.
- ✓ Default T(TILE_SIZE) = 16 for entire experiments.

1. Inside the kernel code, floating operation is occurred here.

```
for (int I = 0; I < N; I += T) {
   ...
  if (row < M && col < K) {
    for (int j = 0; j < T; ++j) {
       p += sA[ty][j] * sB[j][tx];
    }
  }
}
```

For each if branch, floating operation is occurred $2T$ times. Variable `row` and `col` is uniquely determined by each block and thread, and it can have all values less then M, K. Hence, floating operation is performed for only MK threads. Outermost loop is performed $\left\lceil \frac{N}{T} \right\rceil$ times. Therefore, entire floating operation is performed $2MKT\left\lceil \frac{N}{T} \right\rceil = 32MK\left\lceil \frac{N}{16} \right\rceil$ times, where T = 16 for default.

M	16	64	64	112	168	512	1024	1024	4096
N	16	64	128	48	168	510	1024	2048	8000
K	16	64	64	16	168	512	1000	1500	512
#floating operations	8192	524288	1048576	172032	9934848	268435456	2097152000	6291456000	33554432000

2. Inside the kernel code, global read operation is occurred here.

```
for (int I = 0; I < N; I += T) {
  if (row < M && I + tx < N) {
    sA[ty][tx] = A[row * N + I + tx];
  } else {
    sA[ty][tx] = 0;
  }
  if (I + ty < N && col < K) {
    sB[ty][tx] = B[(I + ty) * K + col];
  } else {
    sB[ty][tx] = 0;
  }
...
}
```

Considering read access to `A`, outermost loop is performed $\left\lceil \frac{N}{T} \right\rceil$ times for each thread, but inner if branch restricts `I + tx < N`. This means that the last Iteration of I is accessible only by the remainder divided by T, with a total of N approaches grouped by ThreadX. Variable row can have all values less then M, so the read access A is performed MN times. Likewise, read access to B is performed NK times. Therefore, entire read global memory is occurred N(M + K) times.

M	16	64	64	112	168	512	1024	1024	4096
N	16	64	128	48	168	510	1024	2048	8000
K	16	64	64	16	168	512	1000	1500	512
#floating operations	512	8192	16384	6144	56448	522240	2072576	5169152	36864000

3.Inside the kernel code, global write operation is occurred here.

```
if (row < M && col < K) {
  C[row * K + col] = p;
}
```

This is way more easier than previous questions, global write operation is performed MK times.

M	16	64	64	112	168	512	1024	1024	4096
N	16	64	128	48	168	510	1024	2048	8000
K	16	64	64	16	168	512	1000	1500	512
#floating operations	256	4096	4096	1792	28224	262144	1024000	1536000	2097152

4.

- Use FMA (`__fmaf()`) operation supported in CUDA to increase MAC speed.

- Implementing asynchronous memory access(streaming) and pipelining enables performing copying and calculation simultaneously.

5. This is entire code.

```
...
__global__ void matrixMultiplyShared(float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns) {
  //@@ Insert code to implement matrix multiplication here
  //@@ You have to use shared memory for this lab
  __shared__ float sA[T][T];
  __shared__ float sB[T][T];

  int bx = blockIdx.x, by = blockIdx.y, tx = threadIdx.x, ty = threadIdx.y;
  int row = by * blockDim.y + ty;
  int col = bx * blockDim.x + tx;
  int M = numARows, N = numAColumns, K = numCColumns;
  float p = 0;

  for (int I = 0; I < N; I += T) {
    if (row < M && I + tx < N) {
      sA[ty][tx] = A[row * N + I + tx];
    } else {
      sA[ty][tx] = 0;
    }
    if (I + ty < N && col < K) {
      sB[ty][tx] = B[(I + ty) * K + col];
    } else {
      sB[ty][tx] = 0;
    }
    __syncthreads();

    if (row < M && col < K) {
      for (int j = 0; j < T; ++j) {
        p += sA[ty][j] * sB[j][tx];
      }
    }
    __syncthreads();
  }
  if (row < M && col < K) {
    C[row * K + col] = p;
  }
}

int main(int argc, char **argv) {
  ...
  //@@ Set numCRows and numCColumns
  numCRows    = numARows;
  numCColumns = numBColumns;
```

```
  //@@ Allocate the hostC matrix
  hostC = (float*)malloc(numCRows * numCColumns * sizeof(float));
  gpuTKTime_stop(Generic, "Importing data and creating memory on host");

  gpuTKLog(TRACE, "The dimensions of A are ", numARows, " x ", numAColumns);
  gpuTKLog(TRACE, "The dimensions of B are ", numBRows, " x ", numBColumns);

  gpuTKTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
  cudaMalloc((void**)&deviceA, sizeof(float) * numARows * numAColumns);
  cudaMalloc((void**)&deviceB, sizeof(float) * numBRows * numBColumns);
  cudaMalloc((void**)&deviceC, sizeof(float) * numCRows * numCColumns);

  gpuTKTime_stop(GPU, "Allocating GPU memory.");

  gpuTKTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  cudaMemcpy(deviceA, hostA, sizeof(float) * numARows * numAColumns, cudaMemcpyHostToDevice);
  cudaMemcpy(deviceB, hostB, sizeof(float) * numBRows * numBColumns, cudaMemcpyHostToDevice);

  gpuTKTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here
  dim3 gridSize((numBColumns + T - 1) / T, (numARows + T - 1) / T);
  dim3 blockSize(T, T);

  gpuTKTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here
  matrixMultiplyShared<<<gridSize, blockSize>>>
    (deviceA, deviceB, deviceC, numARows, numAColumns, numBRows, numBColumns, numCRows, numCColumns);

  cudaDeviceSynchronize();
  gpuTKTime_stop(Compute, "Performing CUDA computation");

  gpuTKTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here
  cudaMemcpy(hostC, deviceC, sizeof(float) * numCRows * numCColumns, cudaMemcpyDeviceToHost);

  gpuTKTime_stop(Copy, "Copying output memory to the CPU");

  gpuTKTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here
  cudaFree(deviceA);
  cudaFree(deviceB);
  cudaFree(deviceC);

  gpuTKTime_stop(GPU, "Freeing GPU Memory");

  gpuTKSolution(args, hostC, numCRows, numCColumns);

  free(hostA);
  free(hostB);
  free(hostC);

  return 0;
}
```

6. Tile size is 16, evaluation is performed in NVIDIA RTX A5000, with CUDA 12.0 environment.

M	16	64	64	112	168	512	1024	1024	4096
N	16	64	128	48	168	510	1024	2048	8000
K	16	64	64	16	168	512	1000	1500	512
Importing and allocate on host (ms)	0.275956	1.76676	3.27232	1.52235	10.338	90.1403	355.168	859.101	6144.69
Allocating GPU memory (ms)	0.151149	0.153429	0.159231	0.164408	0.163761	0.273054	0.378074	0.426255	164.413
Copying input to the GPU (ms)	0.035859	0.045739	0.051414	0.040721	0.060956	0.355251	0.904682	1.99645	13.3787
CUDA computation (ms)	0.023414	0.024199	0.025889	0.024329	0.029085	0.157525	1.0272	5.21517	15.744
Copying output from the GPU (ms)	0.014634	0.017536	0.018637	0.015715	0.036302	0.741079	0.484223	3.02174	4.17716
Freeing GPU memory (ms)	0.124431	0.124082	0.128643	0.135004	0.131776	0.250141	0.46387	0.768821	2.47076

7. Evaluation is performed in NVIDIA RTX A5000, with CUDA 12.0 environment.

TILE SIZE	2	4	8	12	16	24	32
Importing and allocate on host (ms)	6116.81	6120.56	6117.27	6136.62	6135.4	6152.58	6131.13
Allocating GPU memory (ms)	10.7291	0.685246	0.687486	0.67873	0.666872	0.682409	0.702006
Copying input to the GPU (ms)	40.7816	40.4968	40.5183	40.5053	40.4436	13.5462	40.6358
CUDA computation (ms)	686.389	106.788	19.9219	37.1805	15.7466	16.8971	17.4443
Copying output from the GPU (ms)	5.19134	4.25415	4.18921	4.68407	4.22715	4.55327	4.66852
Freeing GPU memory (ms)	2.45768	2.48772	2.46737	2.46128	2.46402	2.48867	2.52773