[CSED490C] Assignment Report: Lab2_cuda

- Student Id : 20220848
- Name : 선민수

1. Answering follwing questions

Q: How many floating operations are being performed by your kernel?

A: Total `numAColumns x numBRows x numCRows x numCColumns` times of floating operations are performed. Current kernel does not use pre-computed value or something to reduce the floating point operations.

Q: How many global memory reads are being performed by your kernel?

A: Total `numAColumns x numBRows x numCRows x numCColumns` times of global memory reads. Current kernel needs all the elements for calculating a single element be loaded from global memory.

Q: How many global memory writes are being performed by your kernel?

A: Total `numCRows x numCColumns` times global memory writes. Current Kernel performs global memory writes for only storing the each element in the result matrix.

Q: Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

A: We can hide time for loading the input matrices by starting the threads when sufficient data are loaded into GPU's global memory. For example, we can launch thread block immediately when the sufficent data for calculating the a tile in matrix.

2. Template.cu

```
#include <gputk.h>

#define TILE_WIDTH 32

#define gpuTKCheck(stmt)                                                     \
  do {                                                                    \
    cudaError_t err = stmt;                                               \
    if (err != cudaSuccess) {                                             \
      gpuTKLog(ERROR, "Failed to run stmt ", #stmt);                         \
      gpuTKLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));      \
      return -1;                                                          \
    }                                                                     \
  } while (0)

// Compute C = A * B
__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 ds_A[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;

  float Cvalue = 0.0;

  for (int phase = 0; phase < (numAColumns - 1) / TILE_WIDTH + 1; phase++) {
    if (Row < numCRows && phase * TILE_WIDTH + tx < numAColumns) ds_A[ty][tx] = A[Row *
    else ds_A[ty][tx] = 0.0;

    if (Col < numCColumns && phase * TILE_WIDTH + ty < numBRows) ds_B[ty][tx] = B[(phas
    else ds_B[ty][tx] = 0.0;
    __syncthreads();

    if (Row < numCRows && Col < numCColumns) { for (int ii = 0; ii < TILE_WIDTH; ii++)

    __syncthreads();
  }

  if (Row < numCRows && Col < numCColumns) C[Row * numCColumns + Col] = Cvalue;
}

int main(int argc, char **argv) {
  gpuTKArg_t args;
  float *hostA; // The A matrix
  float *hostB; // The B matrix
  float *hostC; // The output C matrix
  float *deviceA;
  float *deviceB;
  float *deviceC;
  int numARows;    // number of rows in the matrix A
  int numAColumns; // number of columns in the matrix A
  int numBRows;    // number of rows in the matrix B
  int numBColumns; // number of columns in the matrix B
  int numCRows;    // number of rows in the matrix C (you have to set this)
  int numCColumns; // number of columns in the matrix C (you have to set
                   // this)

  args = gpuTKArg_read(argc, argv);

  gpuTKTime_start(Generic, "Importing data and creating memory on host");
  hostA = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 0), &numARows,
                            &numAColumns);
  hostB = (float *)gpuTKImport(gpuTKArg_getInputFile(args, 1), &numBRows,
                            &numBColumns);
  //@@ Set numCRows and numCColumns
  numCRows    = numARows;
  numCColumns = numBColumns;
  //@@ Allocate the hostC matrix here
  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, numARows * numAColumns * sizeof(float));
cudaMalloc((void **)&deviceB, numBRows * numBColumns * sizeof(float));
cudaMalloc((void **)&deviceC, numCRows * numCColumns * sizeof(float));

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

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

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

//@@ Initialize the grid and block dimensions here
dim3 gridSize((numCColumns - 1) / TILE_WIDTH + 1, (numCRows - 1) / TILE_WIDTH + 1, 1)
dim3 blockSize(TILE_WIDTH, TILE_WIDTH, 1);
size_t sharedMemorySize = TILE_WIDTH * TILE_WIDTH * sizeof(float) * 2;

gpuTKTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here
matrixMultiplyShared<<<gridSize, blockSize, sharedMemorySize>>>(deviceA, deviceB, dev

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, numCRows * numCColumns * sizeof(float), cudaMemcpyDeviceTo

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;
}
```

3. Execution times

Execution Systems

All compilation and the executions are made on docker container.

TITANXP

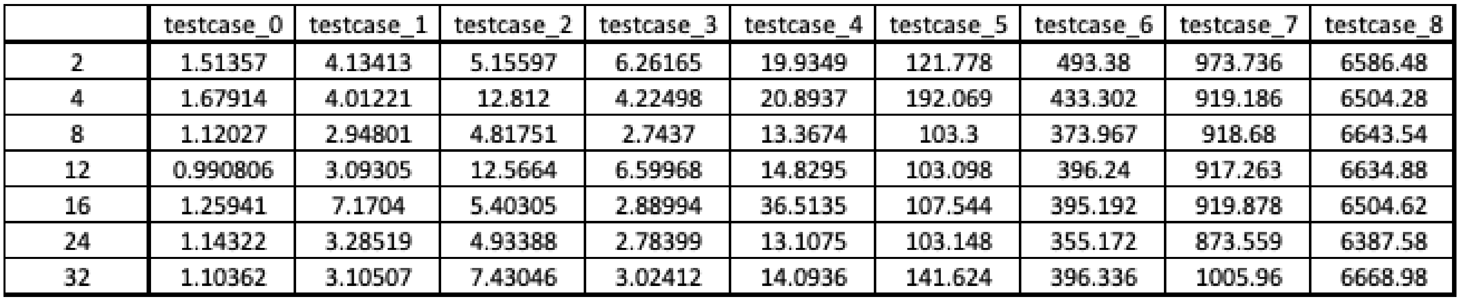
```
srun -p titanxp -N 1 -n 6 -t 02:00:00 --gres=gpu:1 --pty /bin/bash -l
```

- Cluster : `cse-cluster1.postech.ac.kr`
- Docker Image : `nvidia:cuda/12.0.1-devel-ubuntu22.04`
- Driver Version : `525.85.12`
- Cuda Version : `12.0`

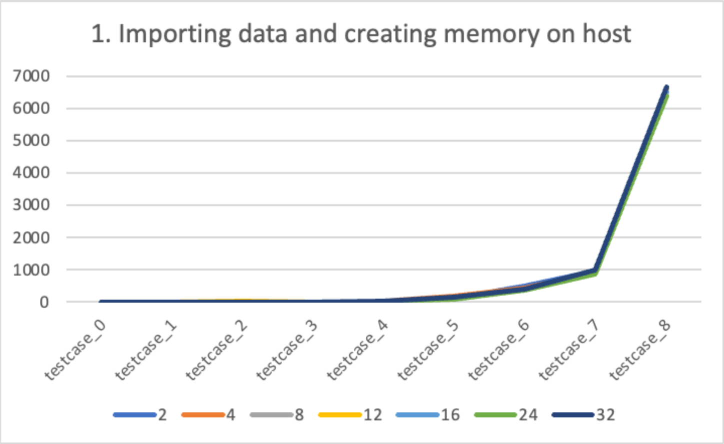
Execution Times

- All the time measurement unit is millisecond(ms).
- Single integer from index names or column names indicates the `TILE_WIDTH` .

1 [Importing data and creating memory on host]

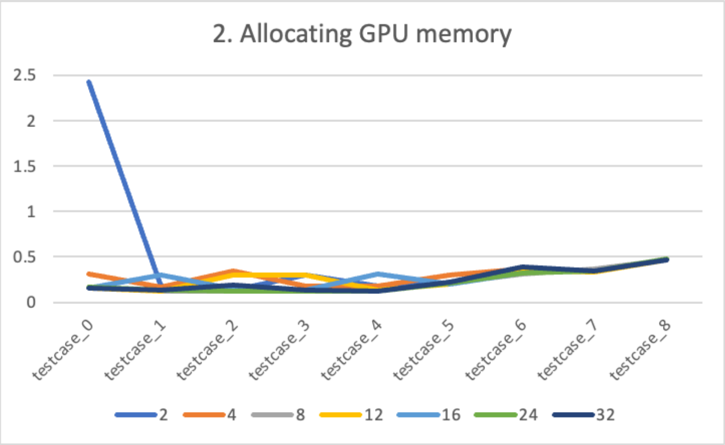


	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	1.51357	4.13413	5.15597	6.26165	19.9349	121.778	493.38	973.736	6586.48
4	1.67914	4.01221	12.812	4.22498	20.8937	192.069	433.302	919.186	6504.28
8	1.12027	2.94801	4.81751	2.7437	13.3674	103.3	373.967	918.68	6643.54
12	0.990806	3.09305	12.5664	6.59968	14.8295	103.098	396.24	917.263	6634.88
16	1.25941	7.1704	5.40305	2.88994	36.5135	107.544	395.192	919.878	6504.62
24	1.14322	3.28519	4.93388	2.78399	13.1075	103.148	355.172	873.559	6387.58
32	1.10362	3.10507	7.43046	3.02412	14.0936	141.624	396.336	1005.96	6668.98



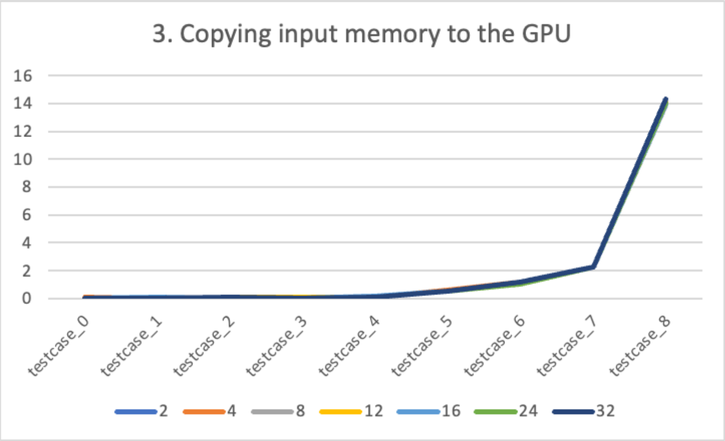
2 [Allocating GPU memory]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	2.43191	0.174405	0.129849	0.301086	0.178674	0.203139	0.358914	0.34374	0.462005
4	0.313372	0.16274	0.338601	0.177887	0.176753	0.299812	0.360922	0.34539	0.465982
8	0.153933	0.121314	0.123831	0.124363	0.127357	0.205958	0.311992	0.366969	0.462135
12	0.157873	0.12737	0.304308	0.297761	0.130018	0.202642	0.345055	0.336682	0.46321
16	0.156764	0.295396	0.13901	0.127951	0.311011	0.204017	0.318304	0.348794	0.468259
24	0.162098	0.131886	0.122963	0.11983	0.122482	0.215762	0.321964	0.347267	0.4751
32	0.155217	0.129834	0.186703	0.138444	0.126708	0.220142	0.388916	0.348535	0.463831

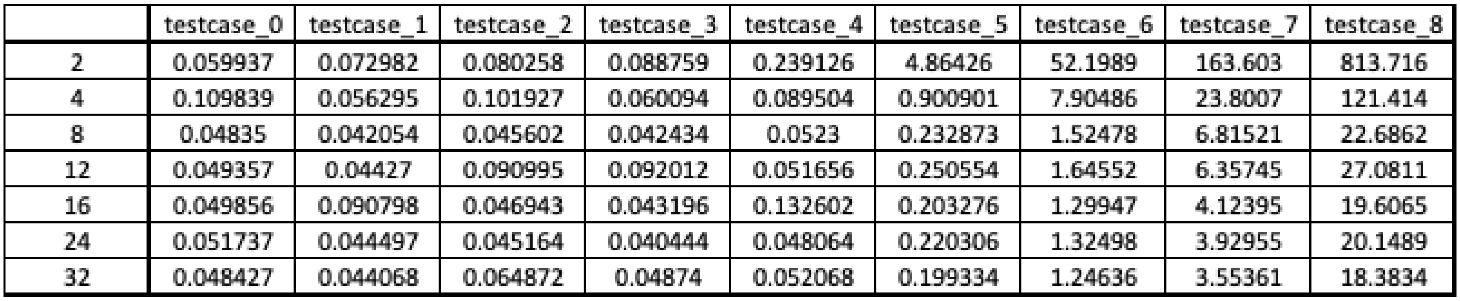


3 [Copying input memory to the GPU]

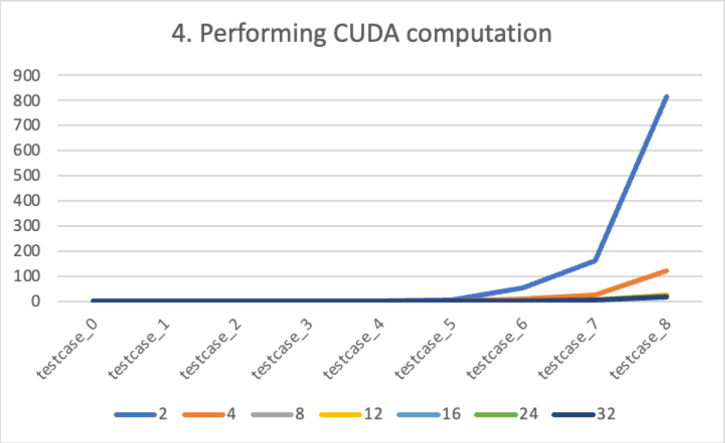
	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	0.055599	0.05891	0.054506	0.079325	0.101959	0.506445	1.17851	2.24743	13.9974
4	0.070943	0.057102	0.110572	0.065453	0.098621	0.593343	1.14758	2.24159	14.1231
8	0.048922	0.047741	0.051859	0.043061	0.08307	0.515585	1.03535	2.24634	13.9124
12	0.047566	0.047104	0.124647	0.079098	0.079541	0.507858	1.14661	2.22977	14.1327
16	0.045692	0.08297	0.05497	0.042962	0.14908	0.503776	1.08464	2.2766	13.9554
24	0.052275	0.04799	0.059776	0.042143	0.077396	0.50618	1.03893	2.2371	14.0606
32	0.048811	0.046776	0.075936	0.047938	0.080259	0.542615	1.16192	2.25854	14.3042



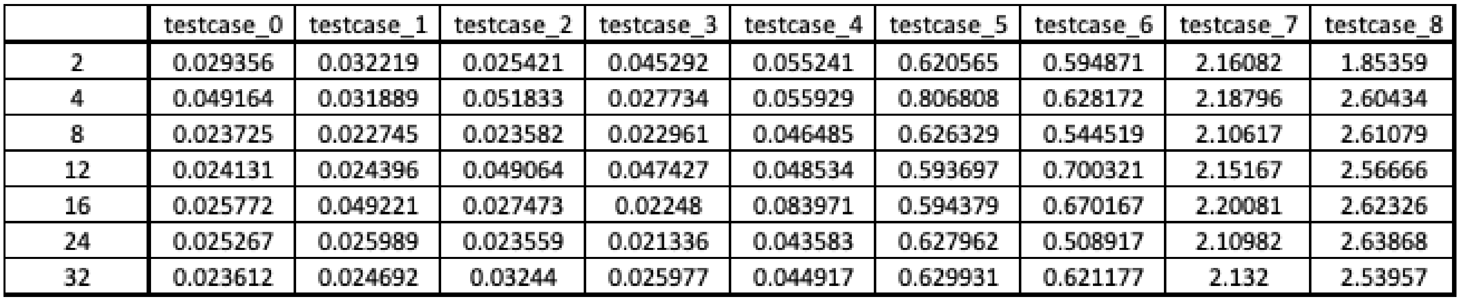
4 [Performing CUDA computation]



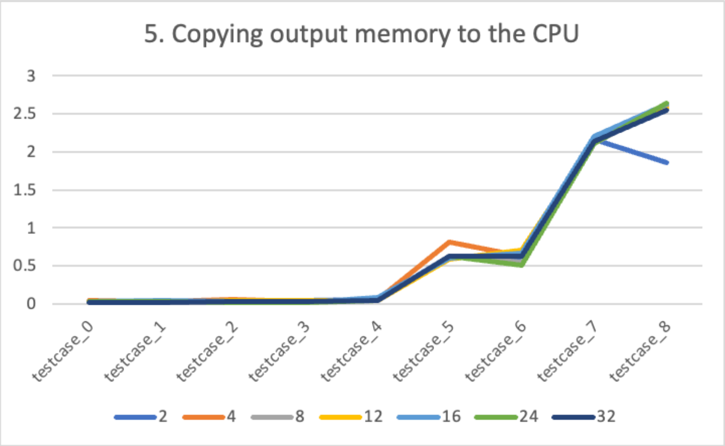
	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	0.059937	0.072982	0.080258	0.088759	0.239126	4.86426	52.1989	163.603	813.716
4	0.109839	0.056295	0.101927	0.060094	0.089504	0.900901	7.90486	23.8007	121.414
8	0.04835	0.042054	0.045602	0.042434	0.0523	0.232873	1.52478	6.81521	22.6862
12	0.049357	0.04427	0.090995	0.092012	0.051656	0.250554	1.64552	6.35745	27.0811
16	0.049856	0.090798	0.046943	0.043196	0.132602	0.203276	1.29947	4.12395	19.6065
24	0.051737	0.044497	0.045164	0.040444	0.048064	0.220306	1.32498	3.92955	20.1489
32	0.048427	0.044068	0.064872	0.04874	0.052068	0.199334	1.24636	3.55361	18.3834



5 [Copying output memory to the CPU]

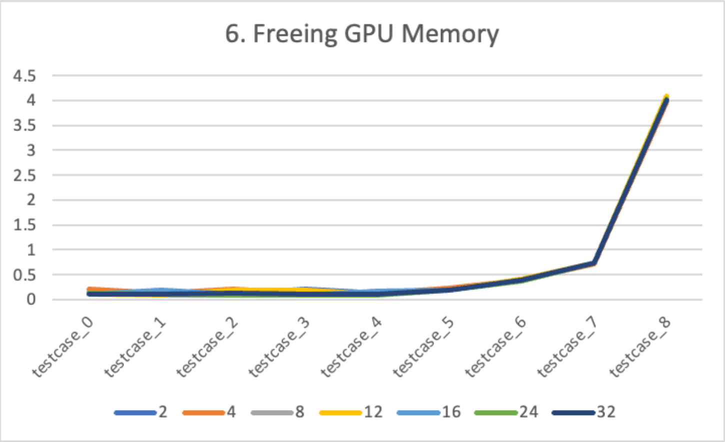


	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	0.029356	0.032219	0.025421	0.045292	0.055241	0.620565	0.594871	2.16082	1.85359
4	0.049164	0.031889	0.051833	0.027734	0.055929	0.806808	0.628172	2.18796	2.60434
8	0.023725	0.022745	0.023582	0.022961	0.046485	0.626329	0.544519	2.10617	2.61079
12	0.024131	0.024396	0.049064	0.047427	0.048534	0.593697	0.700321	2.15167	2.56666
16	0.025772	0.049221	0.027473	0.02248	0.083971	0.594379	0.670167	2.20081	2.62326
24	0.025267	0.025989	0.023559	0.021336	0.043583	0.627962	0.508917	2.10982	2.63868
32	0.023612	0.024692	0.03244	0.025977	0.044917	0.629931	0.621177	2.132	2.53957



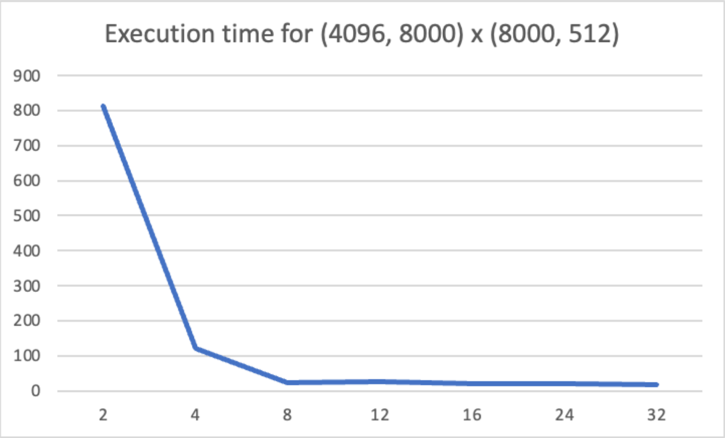
6 [Freeing GPU Memory]

	testcase_0	testcase_1	testcase_2	testcase_3	testcase_4	testcase_5	testcase_6	testcase_7	testcase_8
2	0.152841	0.121445	0.100224	0.201007	0.133892	0.18933	0.398948	0.724678	3.97342
4	0.199539	0.121639	0.203028	0.130663	0.136453	0.236228	0.388899	0.723614	3.97378
8	0.115759	0.094024	0.095776	0.102871	0.099792	0.184729	0.374474	0.733565	4.01729
12	0.111176	0.098242	0.19546	0.188812	0.102984	0.189625	0.40176	0.740305	4.0906
16	0.116621	0.198381	0.10473	0.097422	0.173527	0.187329	0.397935	0.738987	4.00439
24	0.119975	0.102528	0.094011	0.092736	0.096609	0.182549	0.373251	0.733055	4.02037
32	0.115081	0.099305	0.138129	0.104631	0.100314	0.194803	0.39304	0.735244	4.01543



7 [Execution times of the kernel for 4096 * 8000 and 8000 * 512 input matrix]

	2	4	8	12	16	24	32
testcase_8	813.716	121.414	22.6862	27.0811	19.6065	20.1489	18.3834


Execution time for (4096, 8000) x (8000, 512)
900
800
700
600
500
400
300
200
100
0
2
4
8
12
16
24
32