CS 4444 – Report for Homework 5

Jerry Sun (ys7va)

1 Problem Description

The goal of this assignment is to write a traditional matrix multiplication program on GPU using CUDA. The experiment is going to be tested on a K20 GPU on artemis node. The primary goal is to optimize the program on GPU using various optimization techniques.

The formula for this matrix multiplication problem is this:

Assume A is of dimension dim1 * dim2, B is of dimension dim2 * dim3, C should be of dimension dim1 * dim3, then:

$$\forall 0 < i \leq dim1, 0 < j \leq dim3, C_{ij} = \sum_{k=1}^{dim2} A_{ik} * B_{kj}$$

Since it is really time consuming to perform a 10000 * 10000 matrix multiplication using CPU, we only test it using a dimension of 3000 * 3000 within the CPU version of the implementation provided. It takes 93.79s to finish.

2 Result Summary

For $10000 * 10000$ matrices multiplication, my final GPU implementation will take 7.99 seconds in kernel and 9.5 seconds for total runtime, including copy from gloabl memory to device memory. Since I haven't tested the runtime of the CPU version on such a large scale, to make a meaningful comparison, the final optimized version on GPU will only take 0.21420s in kernel execution which is almost 500 times faster then the CPU version. The total runtime is around 1s including memory copy from host to device and device to host. One can expect to see even larger performance gap within a larger problem size.

3 Metadata

3.1 Software

SLURM: version 14.11

nvcc: NVIDIA (R) Cuda compiler driver, V8.0.61

3.2 Hardware

GPU: Tesla K20c, detailed specification can be found here:

https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf

CPU: 2x16c 32c AMD 6276 with 128 GB RAM

4 Approach

For this assignment, it is required to utilize the massive parallelism GPU has to optimize a matrix multiplication problem using CUDA programming model. The general process for this problem works as below:

- malloc necessary memory on device for matrices (both original matrices and result matrieces)
- copy two matrices for multiplication from host to device
- call kernel method and store the result on device memory allocated in first step
- copy the result matrix back onto the host

In CUDA, it performs its parallelism on two levels, block-level and thread-level. We can consider a thread as a mini-task which are grouped along with other threads in blocks. Blocks are grouped in grids and each block can contain up to 1024 threads (or 512 on older GPUs). Each kernel function will launch on every single threads in parallel if possible. In my implementation, each thread will always responsible for calculate one single cell in the final output. More specifically, when calling kernel function it needs to pass into two dim3(special variable in cuda) variable to specify the corresponding block and grid size on which the kernel function will launch. Assume, each thread block is a square with T_block * T_block number of threads. Then, to handle corner cases in which dimension of matrices are not multiple of T_block, I use a ceiling function, the result looks like below:

Note: dim_1, and dim_3 is the row & col number of the final output matrix.

dim3 block(T_block, T_block);

dim3 grid(dim_1 - 1) / T_block + 1, (dim_3 - 1) / T_block + 1)

After successfully divide the whole task into mini-tasks, the key point then is how to implement the kernel function which can cause huge performance difference.

4.1 Baseline Approach

As described above, each thread will be responsible for the computation of one single cell in the final output matrix. The problem we want to figure out then is to retrieve the position of the current thread, since the direct position of the cell relative to the thread in final output matrix can't be directly passed into the kernel function. However, we do have the threadId and blockId as well as the thread/block's dimension prepared inside kernel. Therefore, we can directly map threads on to the matrix, as the position of the threads in the overall grid directly map to the position of the cell in the output matrix. Therefore, we have:

r = BlockId.x * BlockDim.x + ThreadId.x

c = BlcokId.y * BlockDim.y + ThreadId.y

Then the calculation is straightforward as calculating an inner product of one row vector and one column vector:

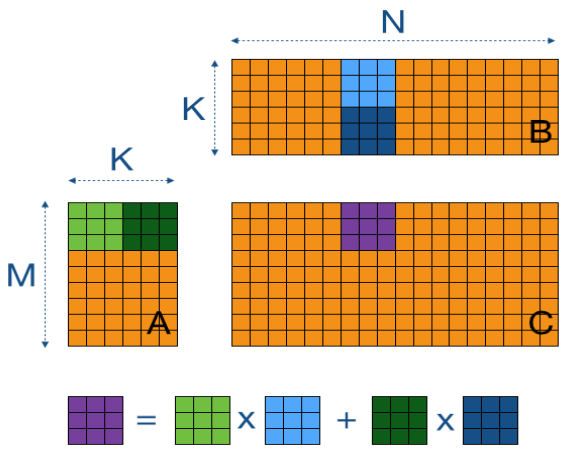
$$C[r][c] = \sum_{i=0}^{dim} A[r][i]B[i][c]$$

4.2 Optimization

One can find that the result of the baseline approach is inefficient from timing result in the next section. This is because, all threads are communicating to the global memory which is super slow. There is one important advantage of GPU, that is the shared memory. The shared memory in CUDA is shared by a single block, that means as long as one thread in a block load the value from global memory to shared memory, other threads can also use the value as long as it is in the same block with the original one.

Therefore we utilize this fact by dividing the whole matrix into tiles instead of directly calculating the inner product of row and column vectors.

More specifically, the idea of utilizing shared memory works as the image below, Note the image is from Cedric Nugteren.



We can find that in tiled memory we multiply corresponding tiles at the same time and finally sum all the producst together.

By doing this, notice that for each block multiplication, each cell in a block is used for multiple times within that block multiplication(more specifically block_size times). Therefore if we preload two sub matrices into shared memory, the overall performance can be benefitted from increasing number of shared memory accesses.

To implement this, it is important to load the correct block into the shared memory, and also handling corner cases become even trickier. Here to simplify the implementation, In my code, the size of a tile is exactly the size of the block.

First we need to divide the whole final matrix into tiles(blocks). Then for each block, it needs to find the corresponding rows in Matrix A, and columns in Matrix B and also divide them into N blocks of the same size(as shown in figure1). Then each thread will iterate through N iterations, so that it can get the sum of each block multiplications and place that value in the spot.

Note, each thread will load exactly two elements from global memory when multiplying two tiles, one is from the element in Matrix A, the other is from element in Matrix B. Also, it is important to synchronize before moving forward into calculation, so that all values are set, otherwise, there might be uninitiated values during calculation.

Further detail implementation can be found directly in code snippets *mult_matrix_shared* in Appendix.

5 Performance

In this section, I will give a brief analysis for the data I retrieved for both versions of kernel function implementation. For time measurement, I use the *start_timer* and *stop_timer* provided in code template. All binary executables are compiled by gcc using -O3 flag on.

For timing, it counts both the total runtime of $computeGPUMMM()$ as well as the kernel function runtime. The program run five times for each number of threads, and I will take the average of that result for further analysis.

5.1 Runtime data collection

Followed are all the timings recorded during testing on Artemis node based on different size of the problem using baseline implementation, all matrices are assumed to be square matrices, and they all use a block size of 32 * 32 if not specified.

Table 1: Baseline with 32 * 32 block

dimension(1000*)	1	2	3	4	5	6	7	8	9	10
Kernel	0.23	1.85	6.27	14.85	29.00	50.13	79.62	118.86	168.48	231.67
Total	1.04	2.65	7.07	15.73	29.89	51.06	80.61	119.97	169.66	232.94

Table 2: Shared Memory with 32 * 32 block

dimension(1000*)	1	2	3	4	5	6	7	8	9	10
Kernel	0.01	0.06	0.22	0.48	1.02	1.73	2.78	3.87	5.93	7.99
Total	0.73	0.87	1.03	1.29	1.91	2.63	3.83	5.00	7.13	9.49

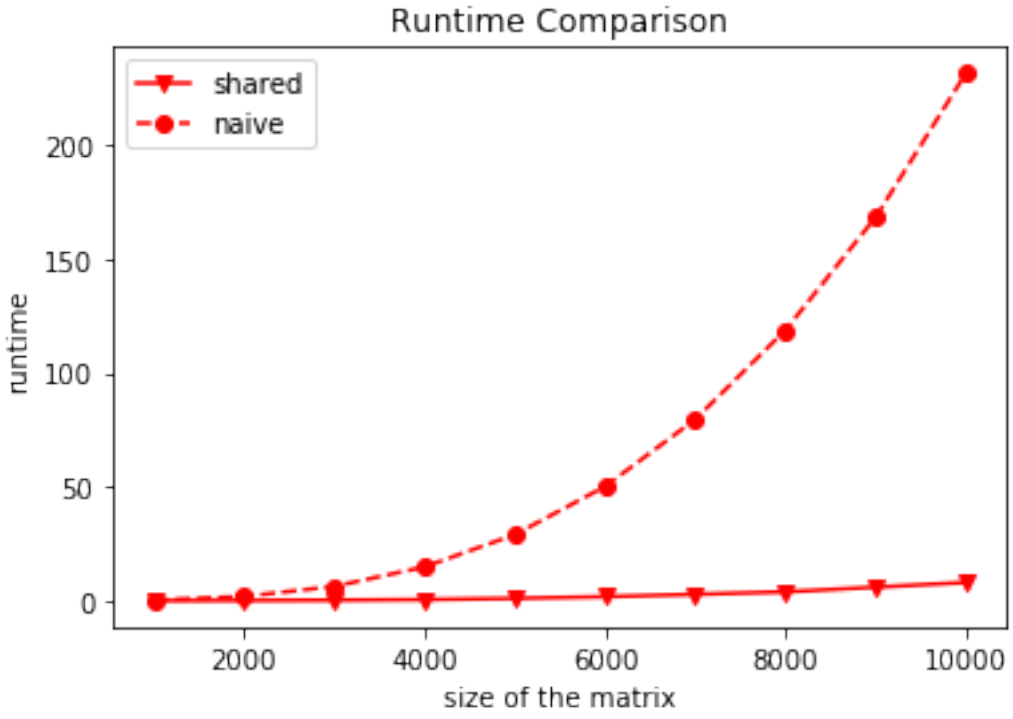
Table 3: Shared Memory with Different Block Size on 10000 * 10000

BLOCK_SIZE	32	16	8	4	2	1
Kernel	7.99	9.68	16.92	75.27	486.55	1273.66
Total	9.49	11.02	18.17	76.58	487.82	1274.96

6 Analysis

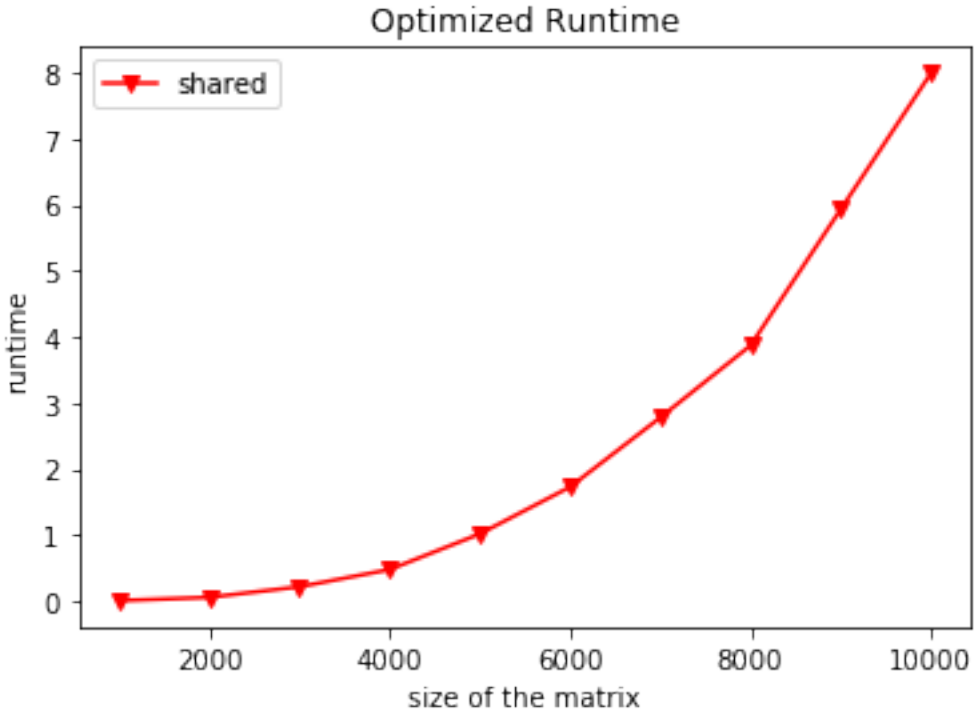
From the data above, we can find that the overhead include memory allocation and transferring takes almost a constant time with fixed size problem, and to get a better sense of the comparison within different kernel functions and block size, we will only use kernel runtime for comparison in this section.

6.1 Shared vs Naive



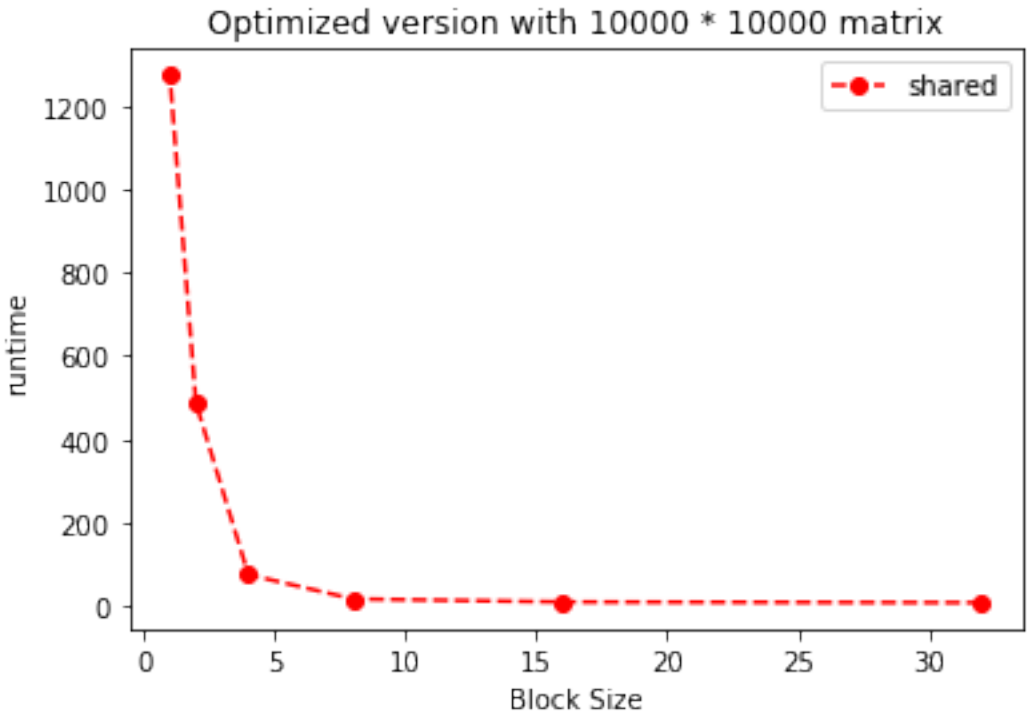
The chart above reveals the kernel runtime comparison with between naive and optimized version. We can found a huge performance increase especially with larger size of the matrix. In fact, with a matrix of size 10000 * 10000, the optimized version is indeed about 30 times faster then the unoptimized version. The reason for this kind of speedup is due to the fact that the access speed for shared memory is much faster then the GPU. By loading two corresponding tiles into shared memory, it can effectively reduce the number of accesses to the global memory. In the unoptimized version, with a matrix size of 10000, each element will be loaded 10000 times. However, in the optimized version, the global memory access will be reduced to around 300 (10000/32) times, which causes a huge performance difference.

6.2 Matrix Size



This chart show the runtime of the optimized kernel function with an increasing size of the problem(matrix size). The curve is pretty smooth but has a larger gradient within a larger size, similar to the one in the last figure. This is understandable since a larger problem size will always cause larger overhead other than computing. Especially in this problem, it has fixed block size of 32, then within a larger problem size, it is indeed increasing the number of blocks initiated. As we know, each block in CUDA is assigned to a multiprocessor, and given a relative small, fixed number of multiprocessors, all other blocks are queued. Within a larger queue, and data size it is common to find a steeper curve with a larger problem size.

6.3 Block Size



Then we want to take a look at how different block size might affect the performance. The chart above shows that when block size is small the performance has a huge difference. The reason is that reducing the block size actually reduce the parallelism the device can provide. As discussed earlier, CUDA will initialize all the blocks onto multiprocessors in a queue which is sequential. The parallelism one can get is from huge number of threads running simultaneously in a single block. Therefore, for example, if the block size is only 4 * 4, then it can only compute 16 cells at a time on a single multiprocessor, which is far fewer than the optimal version.

However, one thing worth notice that, using 16 * 16 block doesn't have a huge performance difference than using 32 * 32 block. There are multiple reasons might cause this effect. One major reason is the possible memory bank conflict. Each multiprocessor divided its shared memory into memory banks. Within each memory bank, the access is sequential, that is being said within a single block, only one thread can access one memory bank at a time, all other threads will be put on a queue, which actually sequentialize the whole process. Within a larger number of blocks, it is highly possible to trigger bank conflict if two threads are accessing continuous shared memory.

7 Conclusion

In this assignment, I have gain solid experience in designing and implementing parallel program using CUDA on GPU. The parallel program I implemented shows a substantial performance speedup which only takes less than 9 seconds to finish a size of 10000 * 10000 matrix multiplication problem. The observation made within two versions of kernel function indicates the influence of memory access, like other problems we have countered, is of huge importance is optimizing problem on GPU.

8 Finally

While this has been a harsh semester for me, I did learn a lot in this class especially in those coding assignment. Thanks for this amazing class.

9 Pledge

On my honor as a student, I have neither received nor given help on this assignment.

Signature: _____

10 Appendix

10.1 matrix_mult.cu

```
//matrix_mult.cu
//template provided by Prof. Andrew Grimshaw
//implementation by Jerry Sun(ys7va) 2017.05.08
//the program will take 4 parameters to specify the size of two matrices
//if only provided 1 value N, it will calculate the multiplication of two N * N matrices
#include<stdio.h>
#include<sys/time.h>
#include<stdlib.h>
#include<iostream>
#include<cuda_runtime.h>
using namespace std;

//Macro to specify block size
#define T_block 32

//---------------------------------- Structures and Globals----------------------------------------
//store dimension of a matrix
typedef struct {
    int dimension1;
    int dimension2;
} ArrayMetadata2D;

// metadata variables describing dimensionalities of all data structures involved in the computation
ArrayMetadata2D A_MD, B_MD, C_MD;
// pointers for input and output arrays in the host memory
// *_CPU is for CPU calculation
// C_GPU_result is for storing GPU calculation result
float *A_CPU, *B_CPU, *C_CPU, *C_GPU_result;
// pointers for input and output arrays in the device memory (NVIDIA DRAM)
float *A_GPU, *B_GPU, *C_GPU;

//---------------------------------- host function definitions ----------------------------------------
void allocateAndInitializeHost();       //allocate and initialize all necessary memory on host machine
void computeCpuMMM();                   //matrix multiplication on CPU
void computeGpuMMM();                   //matrix multiplication on GPU, may use different kernel method
void copyMatricesToGPU();               //copy value in A_CPU & B_CPU to A_GPU & B_GPU respectively
void copyResultFromGPU();               //copy calculated value in C_GPU back into C_GPU_result
void compareHostAndGpuOutput();         //check if the result in C_GPU_result and C_CPU is identical
void die(const char *error);            //end the program
void check_error(cudaError e);          //check memory allocation on cuda
long long start_timer();                //timer for measurement
long long stop_timer(long long start_time, const char *name);  //timer for measurement

//---------------------------------- CUDA function definitions ----------------------------------------
//baseline approach for kernel method, each thread is responsible for one cell in final result
__global__ void mult_matrix_baseline(float *A, float *B, float *C, int dim_1, int dim_2, int dim_3);
```

```
//shared memory version for kernel method, a block of threads read data from DRAM together↩
     into shared
//memory and then do calculation block-wise
__global__ void mult_matrix_shared(float *A, float *B, float *C, int dim_1, int dim_2, int↩
     dim_3);


//↩
    ----------------------------------------------------------------------------------------------------

int main(int argc, char **argv) {
    //parse the command-line argument
    A_MD.dimension1 = (argc > 1) ? atoi(argv[1]) : 100;
    A_MD.dimension2 = (argc > 2) ? atoi(argv[2]) : A_MD.dimension1;
    B_MD.dimension1 = (argc > 3) ? atoi(argv[3]) : A_MD.dimension2;
    B_MD.dimension2 = (argc > 4) ? atoi(argv[4]) : B_MD.dimension1;
    C_MD.dimension1 = A_MD.dimension1;
    C_MD.dimension2 = B_MD.dimension2;

    printf("Matrix A is %d-by-%d\n", A_MD.dimension1, A_MD.dimension2);
    printf("Matrix B is %d-by-%d\n", B_MD.dimension1, B_MD.dimension2);
    printf("Matrix C is %d-by-%d\n", C_MD.dimension1, C_MD.dimension2);
    //if dim2 of A and dim1 of B is different then they can't be multiplied
    if (A_MD.dimension2 != B_MD.dimension1) die("Dimension inconsistent for two matrices")↩
        ;

    //allocate all necessary memory on host
    allocateAndInitializeHost();

    // matrix multiplication in the CPU, commented for large-scale
    // long long CPU_start_time = start_timer();
    // computeCpuMMM();
    // long long CPU_time = stop_timer(CPU_start_time, "\nCPU");

    // matrix multiplication on the GPU
    long long GPU_start_time = start_timer();
    computeGpuMMM();
    long long GPU_time = stop_timer(GPU_start_time, "\tTotal");

    //check the final result
    //commented when CPU result is not available
    //compareHostAndGpuOutput();

    return 0;
}


__global__ void mult_matrix_baseline(float *A, float *B, float *C, int dim_1, int dim_2, ↩
    int dim_3) {
    // retrieve the corresponding row & col in final output matrix
    int r = blockIdx.x * T_block + threadIdx.x;
    int c = blockIdx.y * T_block + threadIdx.y;
    // check if index is in bound
    if (r < dim_1 && c < dim_3) {
        float sum = 0;
        // calculate inner product of two vectors
        for (int i = 0; i < dim_2; i++) {
                sum += A[r * dim_1 + i] * B[i * dim_2 + c];
            }
        // assign final results
        C[r * dim_3 + c] = sum;
    }
}

// Compute C = A * B
```

```
__global__ void mult_matrix_shared(float *A, float *B, float *C, int dim_1, int dim_2, int↩
     dim_3) {

  // store corresponding value in registers
  int b_x = blockIdx.x;
  int b_y = blockIdx.y;
  int t_x = threadIdx.x;
  int t_y = threadIdx.y;

  // retrieve row & col number in final output
  int r = b_y * T_block + t_y;
  int c = b_x * T_block + t_x;

  float s = 0;
  // initiate share memory space
  __shared__ float block_A[T_block][T_block];
  __shared__ float block_B[T_block][T_block];

  // bool variable to check if inbound
  bool inplace = r < dim_1 && c < dim_3;

  // iterate through all blocks in using a ceiling function to deal with corner cases
  for (int m = 0; m < (dim_2 - 1) / T_block + 1; m++) {
    // column num for the retrieved cell in matrix A
    int col = m * T_block + t_x;
    // load value from matrix A, if not available assign 0
    block_A[t_y][t_x] = (r < dim_1 && col < dim_2) ? A[r * dim_1 + col] : 0.0;
    // row num for the retrieved cell in matrix B
    int row = m * T_block + t_y;
    // load value from matrix B, if not available assign 0
    block_B[t_y][t_x] = (row < dim_2 && c < dim_3) ? B[row * dim_3 + c] : 0.0;
    // sync all threads, wait till all threads finish loading
    __syncthreads();

    //if inplace calculate the inner product within two blocks in A and B
    if (inplace)
        for (int i = 0; i < T_block; i++)
            s += block_A[t_y][i] * block_B[i][t_x];
    //sync threads, wait till all threads finish using shared memory in current iteration
    __syncthreads();
  }

  //assign final result
  if (inplace)
    C[r * dim_3 + c] = s;
}

// GPU version MM
void computeGpuMMM() {
    copyMatricesToGPU();
    //for a matrix multiplication problem, only three dimensions are needed
    //two dims for the final matrix, and one for dim2 of A and dim1 of B(identical)
    int dim_1 = A_MD.dimension1;
    int dim_2 = A_MD.dimension2;
    int dim_3 = B_MD.dimension2;
    //initialize gridblock, and threadblock size
    //here we assume each thread always responsible for cell
    dim3 thread(T_block, T_block);
    //if dim_1 not divisible by T_block, we use ceiling function
    //in order to handle corner cases
    dim3 grid((dim_1 - 1) / T_block + 1, (dim_3 - 1) / T_block + 1);
    long long exec_start_time = start_timer();
    //call kernel method, passing in three GPU pointers and three dimensions
    mult_matrix_shared <<<grid, thread>>> (A_GPU, B_GPU, C_GPU, dim_1, dim_2, dim_3);
    //synchroniztion
```

```
    cudaThreadSynchronize();
    stop_timer(exec_start_time, "\tkernal excution time");
    //copy the result from GPU
    copyResultFromGPU();
}



// allocate and initialize A and B using a random number generator,
// also initialize C_CPU and C_GPU_resul
void allocateAndInitializeHost() {
    size_t sizeofA = A_MD.dimension1 * A_MD.dimension2 * sizeof(float);
    A_CPU = (float*) malloc(sizeofA);
    srand(time(NULL));
    for (int i = 0; i < A_MD.dimension1; i++) {
        for (int j = 0; j < A_MD.dimension2; j++) {
            int index = i * A_MD.dimension2 + j;
            A_CPU[index] = (rand() % 1000) * 0.001;
        }
    }

    size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
    B_CPU = (float*) malloc(sizeofB);
    for (int i = 0; i < B_MD.dimension1; i++) {
        for (int j = 0; j < B_MD.dimension2; j++) {
            int index = i * B_MD.dimension2 + j;
            B_CPU[index] = (rand() % 1000) * 0.001;
        }
    }

    size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
    C_GPU_result = (float*) malloc(sizeofC);
    C_CPU = (float*) malloc(sizeofC);

}

// allocate memory in the GPU for all matrices, and copy A and B content from the host CPU↩
     memory to the GPU memory
void copyMatricesToGPU() {
    long long memory_start_time = start_timer();
    size_t sizeofA = A_MD.dimension1 * A_MD.dimension2 * sizeof(float);
    check_error(cudaMalloc((void **) &A_GPU, sizeofA));
    check_error(cudaMemcpy(A_GPU, A_CPU, sizeofA, cudaMemcpyHostToDevice));

    size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
    check_error(cudaMalloc((void **) &B_GPU, sizeofB));
    check_error(cudaMemcpy(B_GPU, B_CPU, sizeofB, cudaMemcpyHostToDevice));

    size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
    check_error(cudaMalloc((void **) &C_GPU, sizeofC));
    stop_timer(memory_start_time, "\nGPU:\tTransfer to GPU");

}

// copy results from C_GPU which is in GPU card memory to C_CPU_result which is in the ↩
    host CPU for result comparison
void copyResultFromGPU() {
    long long memory_start_time = start_timer();
    size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
    check_error(cudaMemcpy(C_GPU_result, C_GPU, sizeofC, cudaMemcpyDeviceToHost));
    stop_timer(memory_start_time, "\tTransfer from GPU");
}

// do a straightforward matrix-matrix multiplication in the CPU
```

```
// notice that this implementation can be massively improved in the CPU by doing proper cache blocking but we are
// not providing you the efficient CPU implementation as that reveals too much about the ideal GPU implementation
void computeCpuMMM() {

    // compute C[i][j] as the sum of A[i][k] * B[k][j] for all columns k of A
    for (int i = 0; i < A_MD.dimension1; i++) {
        int a_i = i * A_MD.dimension2;
        int c_i = i * C_MD.dimension2;
        for (int j = 0; j < B_MD.dimension2; j++) {
            int c_index = c_i + j;
            C_CPU[c_index] = 0;
            for (int k = 0; k < B_MD.dimension1; k++) {
                int a_index = a_i + k;
                int b_index = k * B_MD.dimension2 + j;
                C_CPU[c_index] += A_CPU[a_index] * B_CPU[b_index];
            }
        }
    }
}

// function to determine if the GPU computation is done correctly by comparing the output from the GPU with that
void compareHostAndGpuOutput() {
    int totalElements = C_MD.dimension1 * C_MD.dimension2;
    int missmatchCount = 0;
    for (int i = 0; i < totalElements; i++) {
        if (fabs(C_GPU_result[i] - C_CPU[i]) > 0.01) {
            missmatchCount++;
            printf("mismatch at index %i: %f\t%f\n", i, C_CPU[i], C_GPU_result[i]);
        }
    }
    if (missmatchCount > 0) {
        printf("Computation is incorrect: outputs do not match in %d indexes\n", missmatchCount);
    } else {
        printf("Computation is correct: CPU and GPU outputs match\n");
    }
}

// Prints the specified error message and then exits
void die(const char *error) {
        printf("%s", error);
        exit(1);
}

// If the specified error code refers to a real error, report it and quit the program
void check_error(cudaError e) {
        if (e != cudaSuccess) {
                printf("\nCUDA error: %s\n", cudaGetErrorString(e));
                exit(1);
        }
}

// Returns the current time in microseconds
long long start_timer() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000 + tv.tv_usec;
}


// Prints the time elapsed since the specified time
long long stop_timer(long long start_time, const char *label) {
```

```
    struct timeval tv;
    gettimeofday(&tv, NULL);
    long long end_time = tv.tv_sec * 1000000 + tv.tv_usec;
    printf("%s: %.5f sec\n", label, ((float) (end_time - start_time)) / (1000 * 1000));
    return end_time - start_time;
}
```

10.2 cuda.sh

```
#!/bin/bash
#SBATCH --gres=gpu
#SBATCH --exclusive

# cude.sh
# written by Jerry Sun (ys7va) 2017.05.09
# For submitting jobs through slurm
nvcc -O3 -o matrix matrix_mult.cu
./matrix 10000
```

11 After All

However, I have to say, reports are killing me....