Homework 5: Matrix-Matrix Multiplication with CUDA

Steven Stetzler

Pledge: On my honor as a student, I have neither given nor received unauthorized aid on this assignment.

1 Statement of Completion

I was successfully able to parallelize a matrix multiplication routine for square matrices on a GPU using `CUDA`. In this paper, I examine how run time of matrix multiplication on a GPU scales with the size of the matrix. I also examine how choice of the number of grid blocks and number of threads per block change the run time. We find that using a GPU massively reduces the run time of multiplying large matrices without escaping the $\mathcal{O}(N^3)$ algorithmic run time of matrix multiplication. We also find certain choices of blocking are dramatically better than other choices, indicating that care must be taken to optimize GPU implementation based on both the problem type and the hardware at hand.

2 Problem Description

We seek to parallelize a matrix multiplication routine using a GPU. GPUs are just incredibly parallel computation machines, allow one to start over a thousand threads in an instant, as opposed to the 4 - 8 one might expect to spawn on a modern CPU. Matrix multiplication of a $m \times n$ matrix A and $n \times p$ matrix B to produce a $m \times p$ matrix C is defined as

$$C_{ij} = \sum_{k=0}^{n} A_{ik} \times B_{kj}. \tag{1}$$

The problem of matrix multiplication is susceptible to parallelization because the computation of C_{ij} does not rely on any other value in C. This means that this a data-parallel problem, and can be solved most straightforwardly by having a single computer compute each C_{ij} given the data A_{ik} and B_{kj} for $k \in [0, n)$.

GPUs work by dividing work into blocks and threads. The GPU will run one block at a time, and each block can contain some number of threads to run at once. Getting optimal performance from a GPU requires that that one divides up their problem optimally among threads and blocks.

3 Approach, Results, and Analysis

I wrote a program in `C` that uses calls from the `CUDA` protocol to run computations on a GPU. Throughout, I used the ai nodes on the University of Virginia Computer Science computing cluster to run these tests. The hardware on these nodes follows:

Processor: 4× Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

Cache: 20 MB of Intel SmartCache

Memory: 63 GB of RAM

GPU: Nvidia GeForce GTX 1080 Ti

3.1 Approach

The GPU matrix multiplication routine I used was straightforward: have each thread spawned on the GPU calculate just one value in C. Now, we just need to spawn $N \times N$ threads to compute each value in C, where C is an $N \times N$ matrix. I chose to use a 2 dimensional blocking and threading scheme in order to tackle this problem as matrix multiplication is a naturally 2D problem. I calculated the index (i, j), where i is a row and j is a column of C, that each thread should compute in code as

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Now, all that is left to be done is to choose a block and thread configuration. I found that the maximum number of threads that could be spawned on this GPU was $32 \times 32 = 1024$. Thus, I chose to use a thread block of size 32×32. Then, given an $N \times N$ matrix, the dimension of the block grid required to index all of C is $\lceil \frac{N}{32} \rceil \times \lceil \frac{N}{32} \rceil$.

3.2 Problem Size Scaling

Contained in Fig. 1 is the average run time over 10 trials of the GPU implementation of the matrix multiplication of two $N \times N$ matrices. Figure 1 displays the time to copy the data to the GPU (a significant portion of the total run time), the time taken to perform the matrix multiplication in the GPU, and the sum of the computation time and the copy time.

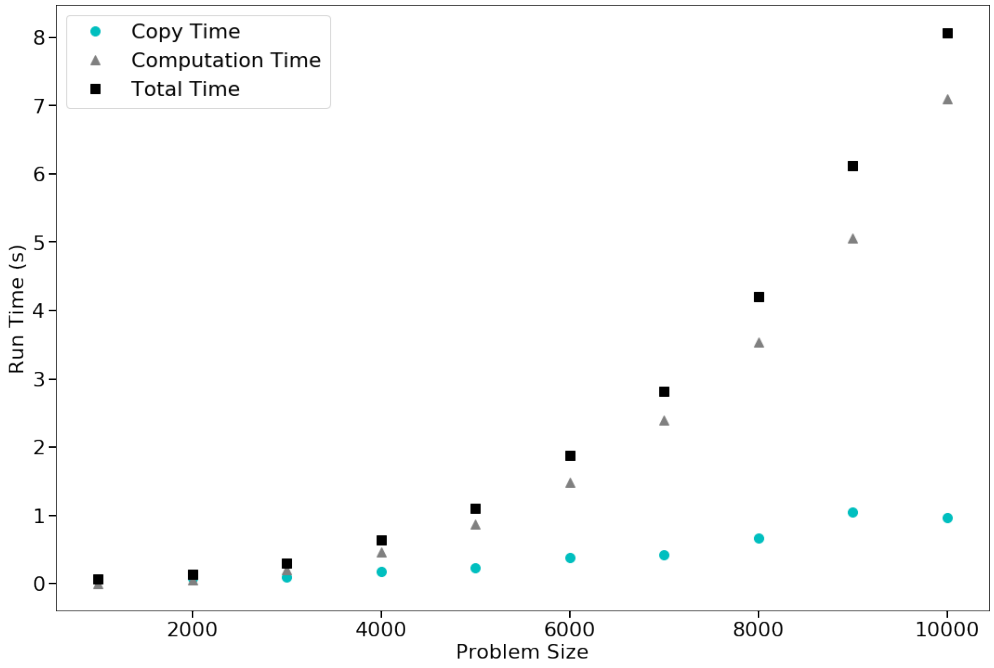

Figure 1: The run time of computing the matrix C in the GPU, of copying memory to and from the GPU, and the combined total run time as a function of the problem size, defined as the dimension N of C.

The data show a power-law scaling of the form

$$T = \alpha \times N^{\beta}$$

where T is the time of computation, N is the problem size, and α and β are fit parameters. We note that the copy time does not show a similar scaling as the computation time scaling. In Fig. 2 we show the data in a log-log scale and we fit the computation time data to get values for α and β. We find a computation

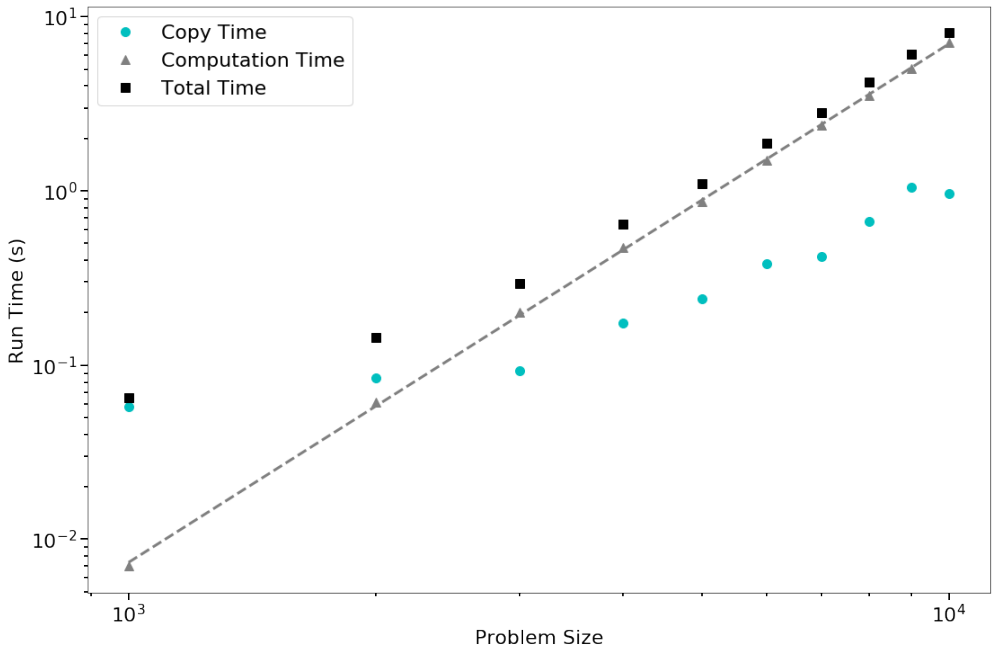

Figure 2: The copy time, computation time, and total run time of the GPU matrix multiplication routine on a log-log scale. Only the computation time shows a power law scaling of $\sim N^3$, which is the algorithmic complexity of the textbook matrix multiply routine.

time scaling of

$$T = 8.6 \times 10^{-12} \times N^{2.98}.$$

This confirms that the scaling of this problem is $\sim \mathcal{O}\left(N^3\right)$, which is the algorithmic complexity of the textbook definition of matrix multiplication.[1] Note that while we didn't change the algorithmic complexity of matrix multiplication in any way, the incredible parallelization of GPUs has lowered the run time cost of matrix multiplication tremendously. In comparison to a naive CPU implementation, this GPU implementation is much faster. For example, with $N = 2000$ a naive CPU matrix multiply (with the processor specifications given above) had a run time of 14.59 (s) as compared to the GPU's run time of 0.043 (s). The GPU implementation is faster than the CPU implementation by a factor of ≈ 337!

3.3 Block and Thread Configurations

I also experimented with how different block and thread configurations changed the run time of the matrix multiplication for $N = 10000$. I continued to use block gridding and thread gridding that was 2 dimensional, as that matched the dimensionality of this problem. I then varied the number of threads in the x direction and the number of threads in a the y direction under the constraint that $T_x \times T_y < 1024$, where T_x and T_y are the number of threads in the x and y dimension respectively. In tweaking the thread gridding, we are automatically tweaking the block gridding as well since the block gridding is necessarily given by $\lceil \frac{N}{T_x} \rceil$ in the x dimension and $\lceil \frac{N}{T_y} \rceil$ in the y dimension. In Fig. 3 I show the run time scaling as a function of T_x given several different values of T_y. In Fig. 4 I show the run time scaling as a function of T_y given several different values of T_x. To differentiate better the run times in each of these figures, I also plot these data on a log-log scale in Fig. 5 and Fig. 6.

These data are interesting because they show that the choice of thread gridding can dramatically affect the run time of the matrix multiplication. The longest run time had $T_x = 1$ and $T_y = 1024$, with a computation time of 153.6 (s). The shortest run time was achieved when $T_x = 16$ and $T_y = 32$ with a computation time

[1] Note that other matrix multiplication algorithms exist that have less than $\mathcal{O}\left(N^3\right)$ scaling. For an example, see https://en.wikipedia.org/wiki/Strassen_algorithm

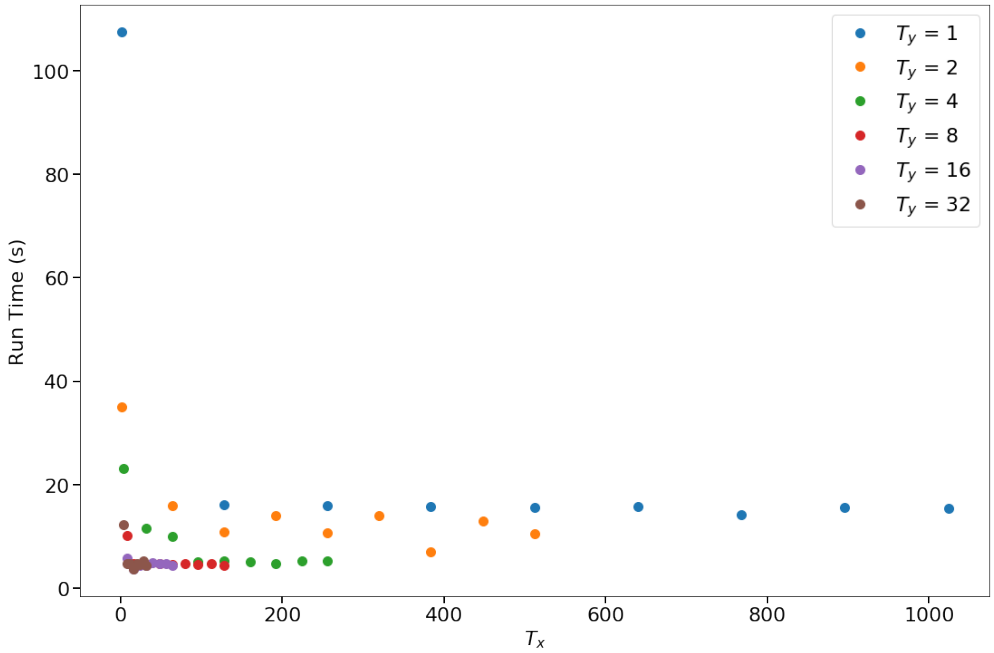

Figure 3: The run time of a GPU implementation of matrix multiplication for $N = 10000$ as function of the thread gridding in the x direction given several choices for T_y

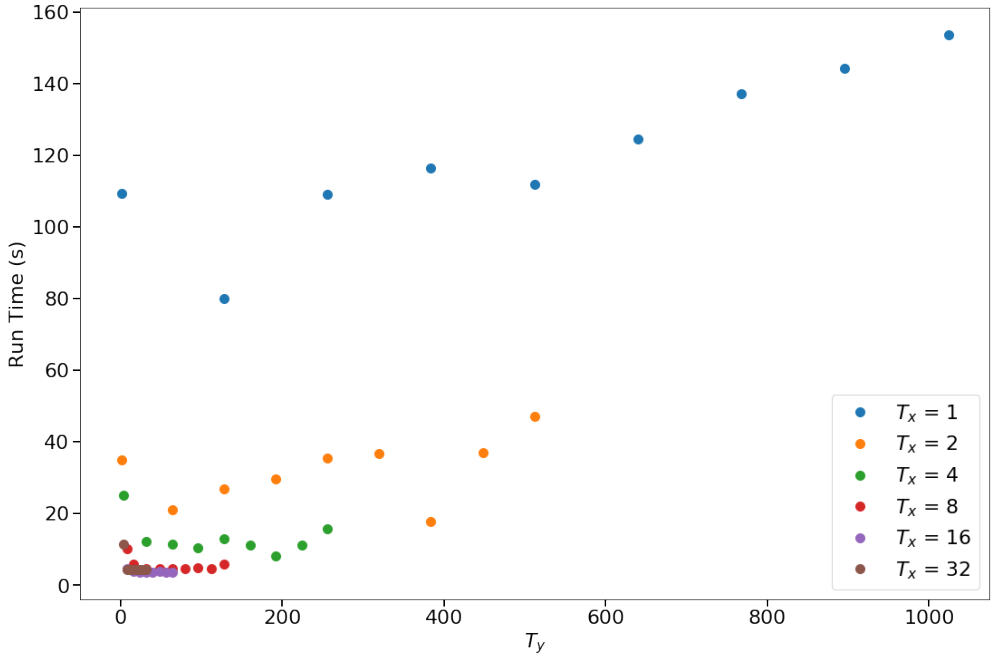

Figure 4: The run time of a GPU implementation of matrix multiplication for $N = 10000$ as function of the thread gridding in the y direction given several choices for T_x

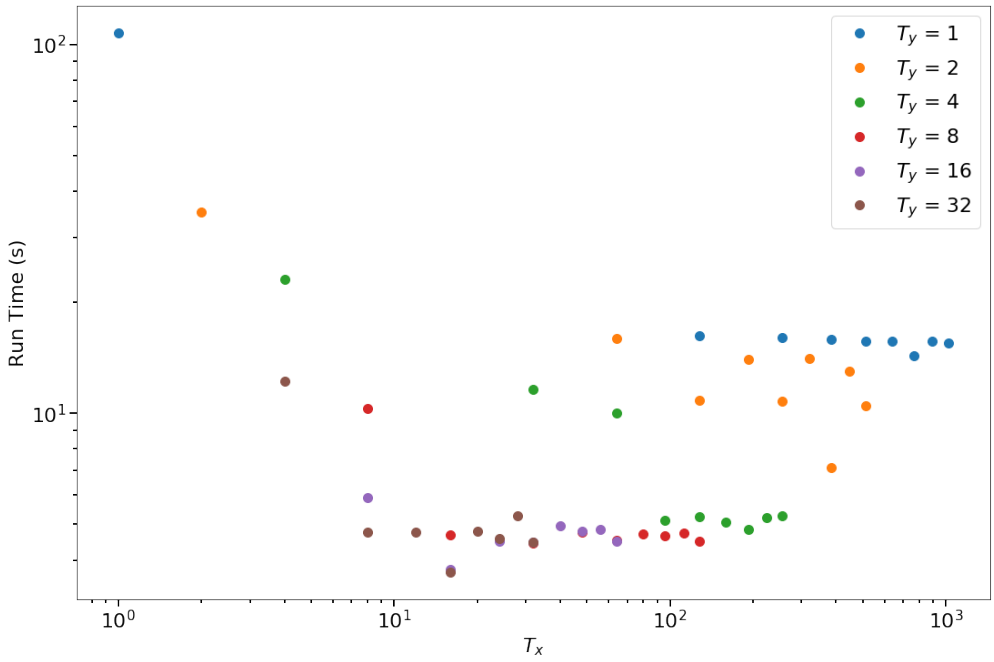

Figure 5: Figure 3 in a log-log scale, differentiating closely packed run times.

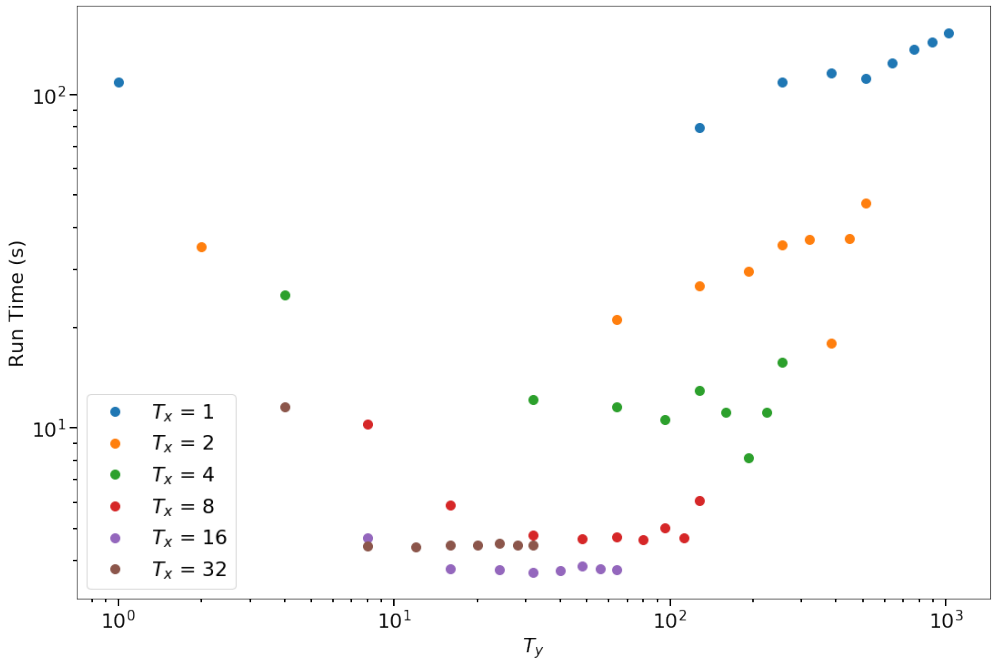

Figure 6: Figure 4 in a log-log scale, differentiating closely packed run times.

of 3.7 (s). The longest run time was $\approx 41.5\times$ slower than the shortest run time! This dramatic difference in run time based on thread gridding occurs due to way thread gridding changes our memory access pattern when performing the matrix multiplication. In choosing $T_x = 1$ and $T_y = 1024$, we are choosing to tackle 1024 rows and only 1 column of C at once. In choosing $T_x = 16$ and $T_y = 32$, we are choosing to tackle 32 rows and 16 columns of C at once. In the optimal case, we would be able to load into the GPU memory all of the data that each block needs to use at once. This requires an optimal memory access pattern. Choosing any extreme value for T_x or T_y will require us to load in a lot of memory that is non-adjacent into the GPU memory (either many of the rows of A or many of the columns of B), thus slowing down our program (as we saw). In choosing moderate values of T_x and T_y, we are able to frequently access memory that is adjacent to one another, as we are loading in fewer rows and columns from A and B, and thus we require fewer memory loads per block. This increases the speed of our program.

4 Conclusion

I was able to successfully implement a matrix multiply routine in a GPU. I found that when multiplying two 2000×2000 matrices, the GPU implementation was faster than a naive CPU implementation by a factor of ≈ 337, providing a tremendously useful speed up for large matrix multiplication. Given the relevance of matrix multiplication, GPU programming with respect to this problem is incredibly important. In tweaking the number of blocks and threads spawned in the GPU to perform the multiplication, I found that some configurations significantly outperformed others. A 2-dimensional block gridding and thread gridding was favorable as setting $T_x = 1$ or $T_y = 1$ gave strictly worse results than all other choices of T_x and T_y. I found that the best performance was achieved with a thread gridding choice of $T_x = 16$ and $T_y = 32$, achieving a $41.5\times$ speed up over the slowest thread gridding choice. In choosing a block and thread gridding for any problem on a GPU, care must be taken to consider the dimensionality of ones problem and the memory access pattern that one expects. The chosen block and thread gridding should optimize the memory access pattern to reduce the amount of times that memory needs to be loaded into the GPU.

Here, I include the GPU implementation of the matrix multiplication.

```
/*
 * CS 4444
 * Steven Stetzler
 * Homework 5: Matrix-Matrix Multiplication with CUDA
 */

#include <stdio.h>
#include <assert.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <sys/time.h>
#include <stdlib.h>
#include <iostream>

using namespace std;

// 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);
    }
}

// A GPU implementation of matrix multiplication.
// Given three N x N matrices A, B, and C we compute C = A x B
```

```
__global__ void matrix_mult_gpu(float* A, float* B, float* C, int N) {
    // Get the row and column of C that this thread should work on
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int k, idx;
    float sum;

    // Ignore threads that would compute values outside of the boundary of the matrix
    if (row < N && col < N) {
        idx = row * N + col;
        sum = 0;
        for (k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[idx] = sum;
    }
}

// A naive (no cache blocking) CPU implementation of matrix multiplication
// Given three N x N matrices A, B, and C we compute C = A x B
void matrix_mult_cpu(float* A, float* B, float* C, int N) {
    int row, col, k, idx;
    float sum;

    for (row = 0; row < N; row++) {
        for (col = 0; col < N; col++) {
            idx = row * N + col;
            sum = 0;
            for (k = 0; k < N; k++) {
                sum += A[row * N + k] * B[k * N + col];
            }
            C[idx] = sum;
        }
    }
}

// Compare the values in two N x N matrices C and C_CPU
void compareHostAndGpuOutput(float* C, float* C_CPU, int N) {
    int totalElements = N * N;
    int missmatchCount = 0;
    for (int i = 0; i < totalElements; i++) {
        if (fabs(C[i] - C_CPU[i]) > 0.01) {
            missmatchCount++;
            printf("mismatch at index %i: %f\t%f\n", i, C[i], C_CPU[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");
    }
}

// Main method
int main(int argc, char** argv) {
    // The problem size N is the dimension of the arrays
    int N = (argc > 1) ? atoi(argv[1]) : 100;
    // Option whether or not to check GPU output against CPU implementation
```

```
// Should not be included for large N, as CPU will be very slow
int check_cpu = (argc > 2) ? atoi(argv[2]) : 0;
// Options to specify the thread gridding
int thread_x = (argc > 3) ? atoi(argv[3]) : 32;
int thread_y = (argc > 4) ? atoi(argv[4]) : 32;
// Option to specify the number of trials for run time tests of the GPU
int n_trials = (argc > 5) ? atoi(argv[5]) : 10;

// Compute block gridding from the thread gridding
// This is the minimum size block gridding given the size of the array and the thread gridding
    that guarantees that all values
// in C will be computed
int grid_x = (int) ceil((double) N / thread_x);
int grid_y = (int) ceil((double) N / thread_y);

// Print run parameters
printf("N = %d\nGrid: %d x %d\nThreads: %d x %d\nTrials: %d\n", N, grid_x, grid_y, thread_x,
    thread_y, n_trials);

// Specify block and thread gridding
dim3 block_per_grid(grid_x, grid_y , 1);
dim3 thread_per_block(thread_x, thread_y, 1);

// Create and allocate three arrays
float* A = (float*) malloc(N * N * sizeof(float));
float* B = (float*) malloc(N * N * sizeof(float));
float* C = (float*) malloc(N * N * sizeof(float));

// Create pointers for GPU arrays, but do not allocate yet!
float* A_GPU;
float* B_GPU;
float* C_GPU;

// Perform random initialization of the arrays
int i, j;
float val;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        val = (rand() % 1000) * 0.001;
        A[i * N + j] = val;
        B[i * N + j] = val;
        C[i * N + j] = val;
    }
}

clock_t start, end;
double elapsed;

// If we want to check against CPU, perform CPU matrix multiplication and time it
if (check_cpu) {
    start = clock();
    matrix_mult_cpu(A, B, C, N);
    end = clock();

    elapsed = (end - start) / (double) CLOCKS_PER_SEC;
    printf("CPU: %.10f seconds\n", elapsed);
}

double copy_avg_time = 0;
```

```
    double comp_avg_time = 0;

    // For each trial in run time analysis
    for (i = 0 ; i < n_trials; i++) {
        // Time the copy operation
        start = clock();

        // Allocate arrays on GPU
        check_error(cudaMalloc((void **) &A_GPU, N * N * sizeof(float)));
        check_error(cudaMalloc((void **) &B_GPU, N * N * sizeof(float)));
        check_error(cudaMalloc((void **) &C_GPU, N * N * sizeof(float)));
        // Copy in values from A, B, and C
        check_error(cudaMemcpy(A_GPU, A, N * N * sizeof(float), cudaMemcpyHostToDevice));
        check_error(cudaMemcpy(B_GPU, B, N * N * sizeof(float), cudaMemcpyHostToDevice));
        check_error(cudaMemcpy(C_GPU, C, N * N * sizeof(float), cudaMemcpyHostToDevice));

        end = clock();

        elapsed = (end - start) / (double) CLOCKS_PER_SEC;
        copy_avg_time += elapsed;

        // Time the computation operation
        start = clock();

        // Perform GPU matrix multiply
        matrix_mult_gpu<<<block_per_grid, thread_per_block>>>(A_GPU, B_GPU, C_GPU, N);
        cudaDeviceSynchronize();

        end = clock();

        elapsed = (end - start) / (double) CLOCKS_PER_SEC;
        comp_avg_time += elapsed;

        // Free arrays if this isn't our last trial
        if (i != n_trials - 1) {
            check_error(cudaFree(A_GPU));
            check_error(cudaFree(B_GPU));
            check_error(cudaFree(C_GPU));
        }
    }

    // Print timing results
    printf("GPU_copy: %.10f seconds\n", copy_avg_time / n_trials);
    printf("GPU: %.10f seconds\n", comp_avg_time / n_trials);

    // If we wanted to check against CPU, do so
    if (check_cpu) {
        // Copy result from GPU
        float* C_GPU_Copy = (float*) malloc(N * N * sizeof(float));
        check_error(cudaMemcpy(C_GPU_Copy, C_GPU, N * N * sizeof(float), cudaMemcpyDeviceToHost));
        // Compare GPU and CPU output
        compareHostAndGpuOutput(C, C_GPU_Copy, N);
    }

    return 0;
}
```