Philip Liberato & Karen Johnson
CS 6444: Parallel Computing
April 22, 2016

Assignment 6: Matrix-Matrix Multiplication with Cuda

Solution Statement

We completed the assignment as specified, satisfying a correct CUDA implementation of the Matrix-Matrix Multiplication problem. For the 10,000 x 10,000 case, our best time achieved on the Rivanna GPUs was 14.44 seconds, with a speedup of 226 over the CPU execution.

Problem Description

The goal of this assignment was to use CUDA to implement a solution to the classic Matrix-Matrix Multiplication problem on a GPU that gives significantly better performance over the traditional CPU solution. Beyond the baseline performance of the naive implementation on the GPU, special work was to be conducted to optimize the performance of our solution on the GPU and provide analysis for the growth of the performance on the GPU relative to the CPU.

Approach and Design

Our implementation for this assignment builds off the structure provided in the mmm_template.cu file and the logic in the vector_add.cu file. The core logic of our solution is contained within several functions, their descriptions are as follows:

- main() - Allocates and initializes the matrices and calculates their product on the CPU and GPU for performance comparison. Contains the logic for padding the A matrix and padding the transpose of the B matrix.
- copyMatricesToGPU() - Copies the padded A and B matrices to the GPU.
- copyResultFromGPU() - Copies the calculated, resultant C matrix from the memory on the GPU back to the memory used by the CPU.

- computeGpuMMM() - Contains the logic for tracking the performance of the GPU implementation as well as getting the data to and from the GPU. The threads per block as well as the grid size are established before launching the computation in the mmm_kernel function.
- mmm_kernel() - Contains the code run on the GPU for multiplying two matrices. The matrices are loaded into shared memory one block at a time.

The previous descriptions detail the logic of the final program. Our solution was iteratively developed to calculate intermediary optimization results. The details of these intermediary stages are described in the next section. The final program was compiled using nvcc with the -O3 optimization flag, and the -arch=sm_30 compilation flag.

Results and Analysis

For this assignment, we elected to run our code on the two machines on Rivanna with GPUs, namely nodes udc-ba30-5 and udc-ba30-7. The specs for these nodes were retrieved using the nvidia-smi command. The nodes used the Tesla K20m graphics card, driver version 331.67 with a warp size of 32. Previous testing was conducted on the Cuda1 machine using a Tesla C2050 card. However, this card did not scale to large matrices as the card would run out of memory. Due to this and the fact that the nodes on Rivanna were newer (and thus faster) all results provided in this document were run on Rivanna.

While working on this assignment, we iteratively completed implementation requirements and recorded the performance results for each addition. Thus, we supply results for analysis on the GPU for a number of cases, each constructively building upon the features of the previous case. Each execution case is described as follows:

- GPU_NAIVE - Implements the naive matrix-matrix multiply solution on the GPU kernel, similar to how the calculation is performed on the CPU.
- GPU_TRANSPOSE_1 - Matrix B is transposed on the GPU and then the naive matrix-matrix multiply solution is run on Matrix A and the transpose of Matrix B in the GPU kernel.
- GPU_TRANSPOSE_2 - Matrix B is transposed on the CPU before the timing of the GPU solution begins. This way, the GPU receives Matrix A and the transpose of Matrix B and doesn't have to perform the transposition before working on the GPU kernel.
- GPU_ROW_PADDING - The rows for Matrix A, the transpose of Matrix B and Matrix C were padded to allow coalescing of reads and writes on the kernel. The padding and subsequent unpadding were done on the CPU outside of the GPU timer.
- GPU_MACHINE_OPT_1 - Uses shared memory on the GPU to load blocks of A and B one at a time in order to implement the block matrix multiply algorithm.
- GPU_MACHINE_OPT_2 - Switches the uses of threadIdx.y and threadIdx.x in the kernel, because we confused the x and y variables and decided to flip them.
- GPU_MACHINE_OPT_3 - Compiled with the -arch=sm_30 flag.

Due to the large time requirement to execute the 10,000x10,000 case on the CPU and its static nature for the purposes of our analysis, we recorded the execution time on the CPU only once and used that to determine the relative speedup for each GPU execution case. Thus for subsequent runs beyond the GPU_NAIVE execution case, the CPU execution was commented out to enable faster execution of our modifications to the file and permit a more flexible assessment of the impact of our changes. Due to the way the time is recorded for performing the matrix-matrix multiplication in the GPU, it is independent of the timing for the CPU execution and thus the removal of the CPU execution had no impact on our timed results. The performance of each execution case and its speedup over the CPU case can be seen in the following table:

Table 1

Matrix-Matrix Multiplication: 10,000 x 10,000 Performance		
Execution Case	**Average Time (s)**	**Average Speedup**
CPU	3265.63	--
GPU_NAIVE	234.77	13.91
GPU_TRANSPOSE_1	234.38	13.93
GPU_TRANSPOSE_2	233.84	13.97
GPU_ROW_PADDING	234.43	13.93
GPU_MACHINE_OPT_1	28.78	113.47
GPU_MACHINE_OPT_2	15.88	205.64
GPU_MACHINE_OPT_3	14.44	226.15

As expected, at each phase in our iterative development process the performance of our implementation improved. The improvements in execution time and speedup can be clearly seen in Figure 1.1 and Figure 1.2 respectively. The block size was manipulated for each of the three optimization phases to assess its impact and to tweak its performance for better results. The best performance achieved for each optimization phase is reported in the previous table. As can clearly be seen, with relatively little effort, our naive implementation produced a substantial speedup over the CPU implementation. While for some purposes this may constitute sufficient performance gains which do not warrant additional optimization effort, we attempted to further improve the performance by tuning the performed computation to be as best suited to the machine as possible. As can be seen, these efforts paid off and resulted in a substantial speedup beyond our naive implementation. Interestingly, transposing B and padding the matrices initially did not result in a significant performance improvement. This is because the benefit of these optimizations was constrained by our continued use of global memory and required the use of shared memory, which wasn't considered until the next experiment in Table 2.

However, implementing shared memory in OPT_1 significantly improved the overall runtime, since accessing global memory is 100 times slower than accessing shared memory. In fact, the speedup of OPT_1 compared to GPU_ROW_PADDING is nearly 100 times faster. From there, flipping the x and y variables in OPT_2 further increased performance as we achieved a better indexing order, causing more memory accesses to be coalesced. Lastly, OPT_3 slightly improved performance by compiling with the -arch=sm_30 flag, which Yan advised us to try. Using this flag causes the code to be compiled for newer GPUs, making it less backwards compatible, but potentially more efficient on newer GPUs.

Figure 1.1

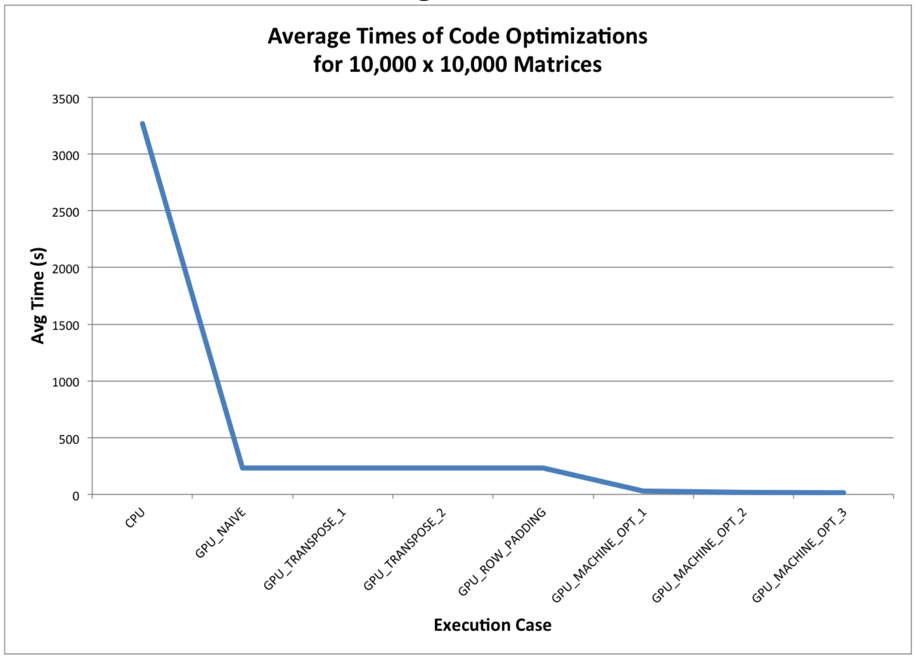
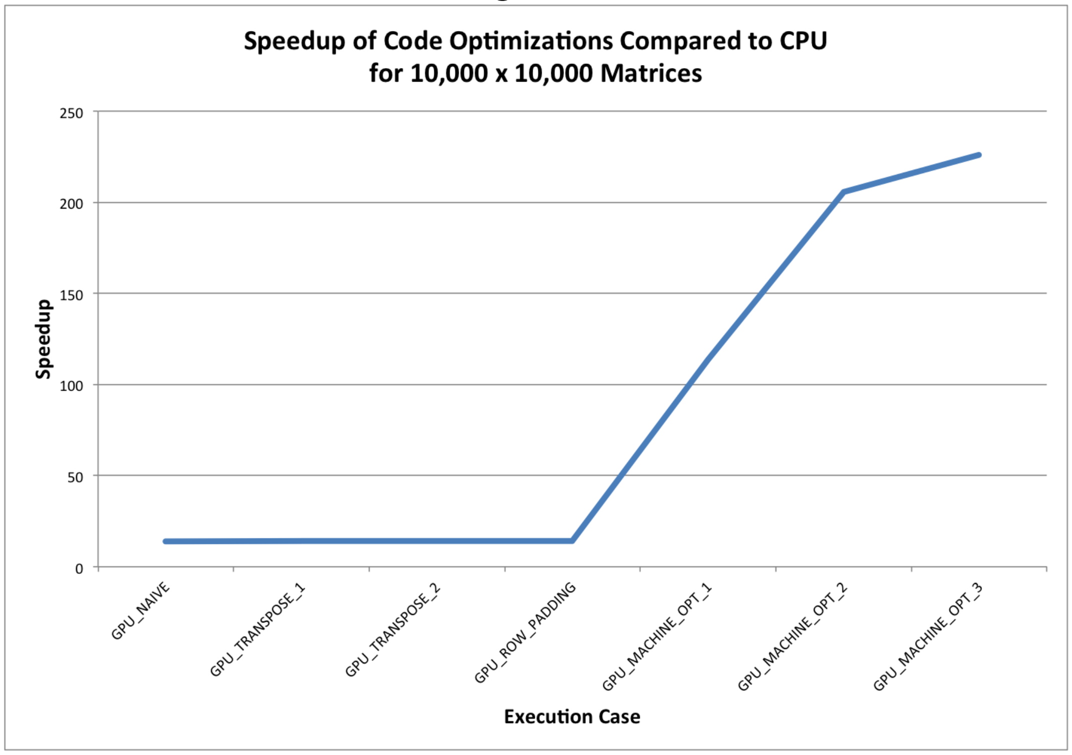


Figure 1.2



To better assess the scalability of our implementation at the various development phases, we produced a chart with the performance time and associated speedup for 10 different square matrix sizes in increments of 1000. The tested cases were OPT_1 with a block dimension of 16x16 and OPT_3 with block dimensions of 8x8 and 16x16. The results are given in Table 2 as follows:

Table 2

MMM: Rivanna with Shared Memory							
Matrix Size	**Average CPU Time (s)**	**Average GPU Time (s)**			**Average Speedup**		
		OPT_ 1 16 x 16	**OPT_3 8 x 8**	**OPT_3 16 x 16**	**OPT_1 16 x 16**	**OPT_3 8 x 8**	**OPT_3 16 x 16**
1,000	1.57	0.85	0.98	1.1	1.85	1.60	1.43
2,000	11.62	1.20	0.96	1.20	9.68	12.10	9.68
3,000	73.72	1.81	1.43	1.37	40.73	51.55	53.81
4,000	n/a	2.80	2.19	1.96	n/a	n/a	n/a
5,000	n/a	4.74	3.15	2.86	n/a	n/a	n/a
6,000	n/a	6.99	4.81	4.08	n/a	n/a	n/a
7,000	n/a	10.69	6.82	5.74	n/a	n/a	n/a
8,000	n/a	15.42	9.49	7.88	n/a	n/a	n/a
9,000	n/a	21.54	13.29	10.91	n/a	n/a	n/a
10,000	3265.63	28.78	17.68	14.44	113.47	184.71	226.15

As expected, OPT_3 - the final optimization phase - produced the best results with the 16x16 block dimensions. This case not only produced the best final result, but consistently scaled the best once overtaking the other test cases beyond the 2,000 matrix size run. It is also clear that using a block size of 16x16 outperforms a block size of 8x8 for every matrix size with OPT_3, this is most likely because there are more threads per block with 16x16, and more work is done more efficiently on the GPU. This result is furthered visualized by plotting the average times of the experiment in Figure 2.1 and the average speedups of the experiment in Figure 2.2.

Figure 2.1

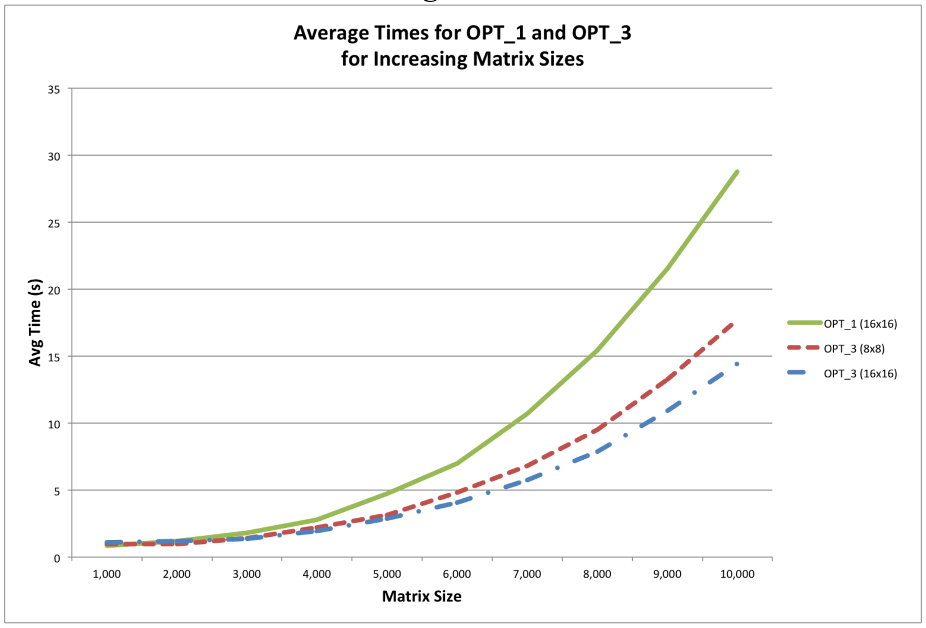
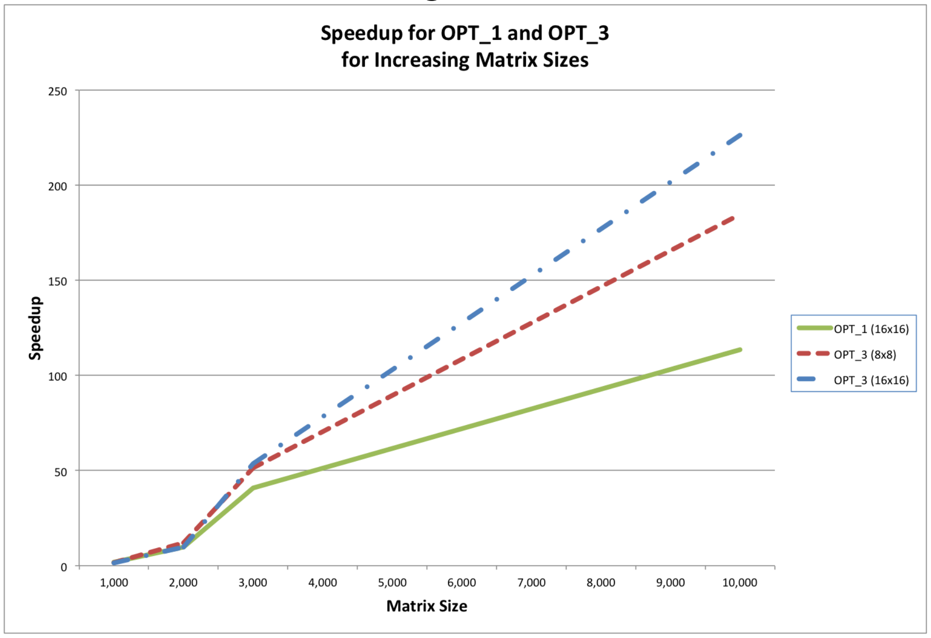


Figure 2.2



A more extensive experiment was conducted on the OPT_1 and OPT_3 optimization phases for 10,000 x 10,000 matrices to tweak the implementation to complement the machine architecture and analyze the impact the block size has on performance. These results are presented in Table 3 and then plotted to visualize the average times in Figure 3.1 and average speedups in Figure 3.2. As previously mentioned, the best results were obtained from the OPT_3 optimization phase with a block size of 16x16. We speculate that these block dimensions produced the best results for OPT_3 and nearly for OPT_1 because the size in relation to the warp size and overall problem size complemented not only the problem but the cuda machines on Rivanna. It follows from this result that other block dimensions similar in size to the 16x16 case would also produce quite good performance. As expected, the 14x14 case nearly matches the OPT_3 phase and also slightly outperforms the OPT_1 phase. As the block size diminishes, the performance begins to drop - and then does so dramatically for the 4x4 case where the block dimensions are only half the warp size.

Table 3

MMM: Rivanna with Shared Memory For 10,000 x 10,000				
Block Size	**Average GPU Time (s)**		**Average Speedup**	
	OPT_1	**OPT_3**	**OPT_1**	**OPT_3**
4 x 4	93.39	88.4	34.97	36.94
8 x 8	23.37	17.67	139.74	184.81
10 x 10	28.17	19.17	115.93	170.35
14 x 14	28.29	14.97	115.43	218.14
16 x 16	28.78	14.44	113.47	226.15
32 x 32	37.35	19.28	87.43	169.38

Figure 3.1

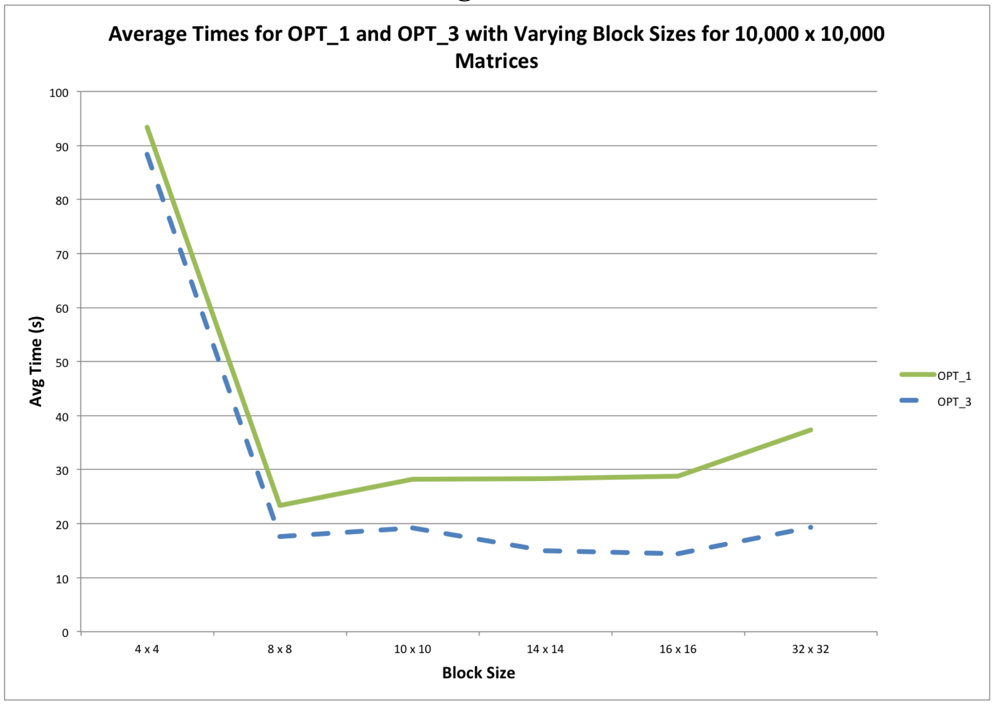
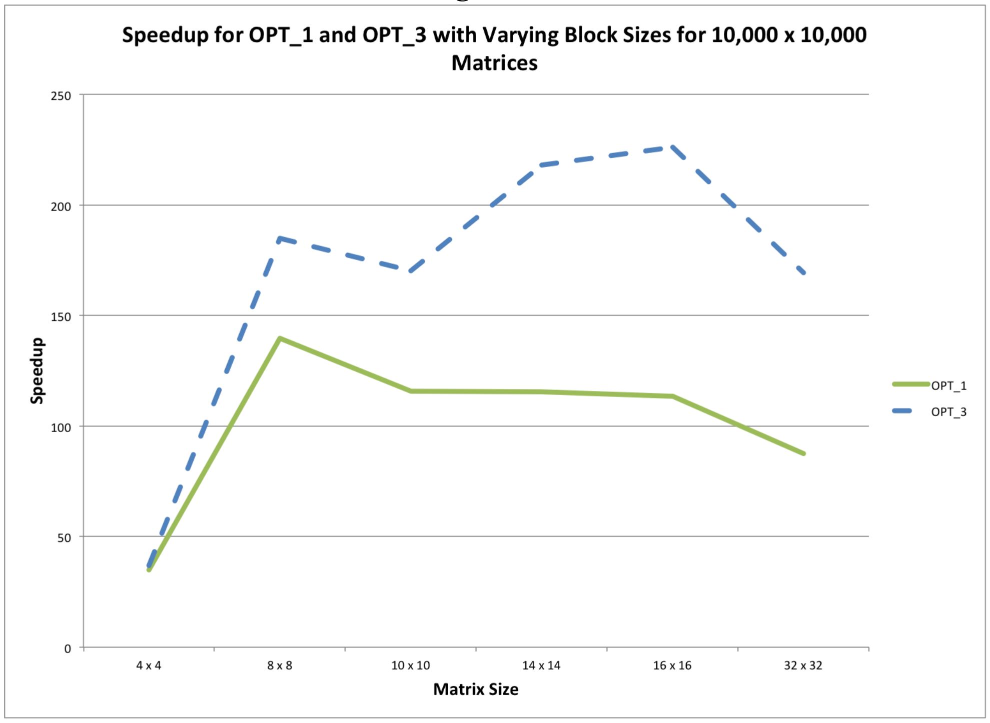


Figure 3.2



Conclusion

In conclusion, our final solution extremely improved upon not only the CPU implementation for matrix-matrix multiplication, but significantly improved upon the naive GPU solution. This assignment constituted a powerful introduction to CUDA programming, which not only familiarized us with the syntax and semantics, but also how to think about tweaking one's code to significantly improve performance by exploiting knowledge of the underlying hardware. The iterative development approach we took to this assignment and overall analysis was enlightening and will inform our future programming endeavors on both the CPU and GPU alike.

On our honor as students, we have neither given nor received aid on this assignment.

Philip & Karen

run.sh

```
#!/bin/bash

# Grab dimensions from command line
dims=$1

# Export dimensions so execute.sh can use them
export dims

# Get the compiler
module load cuda

# Re-compile the program
nvcc -arch=sm_30 -O3 mmm.cu -o mmm

# Submit job
sbatch execute.sh
```

execute.sh

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=00:05:00
#SBATCH --partition=training
#SBATCH --account=parallelcomputing

# Change the below values to be for udc-ba30-5 or -7
#SBATCH --nodelist=udc-ba30-5
#SBATCH --job-name=test5
#SBATCH --output="out/out_cuda5.txt"

# Execute using parameters exported from run.sh
./mmm $dims $dims $dims $dims
```

mmm.cu

```
// mmm.cu
// Guide used: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

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

using namespace std;

//---------------------------------- Structures and
Globals--------------------------------------------

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
```

```
float *A, *B, *C, *C_CPU, *B_TRANS;
// Pointers for input and output arrays in the device memory (NVIDIA DRAM)
float *A_GPU, *B_GPU, *C_GPU;
// Pointers for padded A and B, and unpadded C_RES, for the GPU final result
float *A_PAD, *B_PAD, *C_RES;

// The dimensions of the block
const int BLOCK_SIZE_X = 16;
const int BLOCK_SIZE_Y = 16;

int pad_x;
int pad_y;

//----------------------------------- Host Function Definitions
-----------------------------------------

void allocateAndInitializeAB();
void computeCpuMMM();
void computeGpuMMM();
void copyMatricesToGPU();
void copyResultFromGPU();
void compareHostAndGpuOutput();
void die(const char *error);
void check_error(cudaError e);

//---------------------------- CUDA Function Definitions -----------------------------------
__global__ void mmm_kernel(float *A, float *B, float *C, int Ax, int By);

//---------------------------------------- CODE -------------------------------------------

int main(int argc, char **argv) {

       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);

       allocateAndInitializeAB();

       // Matrix matrix multiplication in the CPU
       clock_t start = clock();
       computeCpuMMM();
       clock_t end = clock();
       double elapsedCPU = (end - start) / (double) CLOCKS_PER_SEC;
       printf("Computation time in the CPU: %f seconds\n", elapsedCPU);

       //---------- MY ADDED STUFF ----------//
       // Transpose B
       size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
       B_TRANS = (float*) malloc(sizeofB);

       // generate trans of B
       for (int j = 0; j < B_MD.dimension2; ++j) {
        for (int i = 0; i < B_MD.dimension1; ++i) {
           B_TRANS[i + j * B_MD.dimension1] = B[j + i * B_MD.dimension1];
        }
    }
    // we could switch B's dimensions, but since we only ever input square matrices, we don't
care :)

    // Pad A and B
    // We know they're the same size so do it at the same time
    int Ax = A_MD.dimension1;
    int count = Ax;
    while (count % BLOCK_SIZE_X != 0) {
        count++;
    }
    pad_x = count - Ax;

    // added this for non square blocks
    int Ay = A_MD.dimension2;
    int count_y = Ay;
    while (count_y % BLOCK_SIZE_Y != 0) {
        count_y++;
    }
    pad_y = count_y - Ay;

    // Allocate PAD arrays
    size_t sizeofPADs = (A_MD.dimension1 + pad_x) * (A_MD.dimension2 + pad_y) * sizeof(float);
        A_PAD = (float*) malloc(sizeofPADs);
        B_PAD = (float*) malloc(sizeofPADs);

        int rowcount = 0;
        int padindex = 0;
        int p = 0;
    for (int i = 0; i < A_MD.dimension1 * A_MD.dimension2; i++) {
        // Do padding because we're at the end of a row
        if (rowcount == Ax) {
                // Add however much padding there is
                while (p < pad_x) {
                        A_PAD[padindex] = 0.0;
                        B_PAD[padindex] = 0.0;
                        p++;
                        padindex++;
                }
                p = 0;
                rowcount = 0;
                i--;
        }
        else {
                A_PAD[padindex] = A[i];
                B_PAD[padindex] = B_TRANS[i];
                padindex++;
                rowcount++;
            }
    }
    // Add however many rows of padding we need
    for (int i = A_MD.dimension2; i < (A_MD.dimension2 + pad_y); i++) {
        for (int j = 0; j < (A_MD.dimension1 + pad_x); j++) {
                A_PAD[i * (A_MD.dimension1 + pad_x) + j] = 0.0;
                B_PAD[i * (B_MD.dimension2 + pad_x) + j] = 0.0;
        }
    }

        // MMM on the GPU
        start = clock();
        computeGpuMMM();
        end = clock();
        double elapsedGPU = (end - start) / (double) CLOCKS_PER_SEC;
        printf("Computation time in the GPU: %f seconds\n", elapsedGPU);

        Compute the speedup or slowdown
        if (elapsedGPU > elapsedCPU) {
                printf("\nCPU outperformed GPU by %.2fx\n", (float) elapsedGPU / (float)
elapsedCPU);
        } else {
                printf("\nGPU outperformed CPU by %.2fx\n", (float) elapsedCPU / (float)
elapsedGPU);
        }

        // Copy C_CPU to a smaller C_RES

        // Allocate C_RES to be normal size of C
        size_t sizeofCRES = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
        C_RES = (float*) malloc(sizeofCRES);

        int offset = 0;
        for (int i = 0; i < C_MD.dimension1 * C_MD.dimension2; i++) {
                if (i % C_MD.dimension1 == 0 && i != 0) {
                        offset += pad_x;
                }
                C_RES[i] = C_CPU[i + offset];
        }

        // Check the correctness of the GPU results
        /compareHostAndGpuOutput();

        return 0;
}

// Allocate and initialize A and B using a random number generator
void allocateAndInitializeAB() {

        size_t sizeofA = A_MD.dimension1 * A_MD.dimension2 * sizeof(float);
        A = (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[index] = (rand() % 1000) * 0.001;
                }
        }

        size_t sizeofB = B_MD.dimension1 * B_MD.dimension2 * sizeof(float);
        B = (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[index] = (rand() % 1000) * 0.001;
                }
        }
}

// 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() {
	size_t sizeofA = (A_MD.dimension1 + pad_x) * (A_MD.dimension2 + pad_y) * sizeof(float);
	check_error(cudaMalloc((void **) &A_GPU, sizeofA));
	check_error(cudaMemcpy(A_GPU, A_PAD, sizeofA, cudaMemcpyHostToDevice));

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

	size_t sizeofC = (C_MD.dimension1 + pad_x) * (C_MD.dimension2 + pad_y) * sizeof(float);
	check_error(cudaMalloc((void **) &C_GPU, sizeofC));
}

// Copy results from C_GPU which is in GPU card memory to C_CPU which is in the host CPU for
result comparison
void copyResultFromGPU() {
	size_t sizeofC = (C_MD.dimension1 + pad_x) * (C_MD.dimension2 + pad_y) * sizeof(float);
	C_CPU = (float*) malloc(sizeofC);
	check_error(cudaMemcpy(C_CPU, C_GPU, sizeofC, cudaMemcpyDeviceToHost));
}

// 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() {

	// Allocate the result matrix for the CPU computation
	size_t sizeofC = C_MD.dimension1 * C_MD.dimension2 * sizeof(float);
	C = (float*) malloc(sizeofC);

	// 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[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[c_index] += A[a_index] * B[b_index];
			}
		}
	}
}

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

```
        } 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);
        }
}

// kernel code
__global__ void mmm_kernel(float *A, float *B, float *C, int Ax, int By) {

       // Store blocks in shared memory
       __shared__ float Ashared[BLOCK_SIZE_X][BLOCK_SIZE_Y];
       __shared__ float Bshared[BLOCK_SIZE_X][BLOCK_SIZE_Y];

       // Get submatrix of C
       float *subC = &C[(Ax * BLOCK_SIZE_Y * blockIdx.y) + (BLOCK_SIZE_X * blockIdx.x)];

       float sum = 0.0;

       // For each submatrix in A/B
       // Multiply subA subB i,j and add to sum
       int num_blocks = Ax / BLOCK_SIZE_X;

       int rAxc = threadIdx.y * Ax + threadIdx.x;
       int rByc = threadIdx.y * By + threadIdx.x;

       float *subA, *subB;


       for (int s = 0; s < num_blocks; s++) {
              // Get submatrix of A
              subA = &A[(Ax * BLOCK_SIZE_Y * blockIdx.y) + (BLOCK_SIZE_X * s)];

              // Get submatrix of B
              subB = &B[(By * BLOCK_SIZE_Y * blockIdx.x) + (BLOCK_SIZE_X * s)];

              // warp loads values
           Ashared[threadIdx.y][threadIdx.x] = subA[rAxc];
           Bshared[threadIdx.y][threadIdx.x] = subB[rByc];

        // Synch to make sure everything is loaded before doing any computation
        __syncthreads();

        // Multiply this thread's row and column together
        for (int k = 0; k < BLOCK_SIZE_X; k++) {
            sum += Ashared[threadIdx.y][k] * Bshared[threadIdx.x][k];
        }

        // Synch again so that everyone is done
        __syncthreads();
       }

       // Write subC back out
       subC[rAxc] = sum;
}

// DO IT TO IT
// MMM on GPU
void computeGpuMMM() {

       // Transfer input to GPU
       clock_t start = clock();
       copyMatricesToGPU();
       clock_t end = clock();
       double elapsed = (end - start) / (double) CLOCKS_PER_SEC;
       printf("GPU: Transfer to GPU: %f seconds\n", elapsed);

       // Execute the kernel to compute the vector sum on the GPU
       start = clock();

       dim3 threadsPerBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
       dim3 numBlocks((A_MD.dimension1 + pad_x) / threadsPerBlock.x, (B_MD.dimension2 + pad_y)
/ threadsPerBlock.y);
       mmm_kernel <<<numBlocks, threadsPerBlock>>> (A_GPU, B_GPU, C_GPU, A_MD.dimension1 +
pad_x, B_MD.dimension2 + pad_y);

       // Make the CPU main thread wait for the GPU kernel call to complete
       cudaThreadSynchronize();  // This is only needed for timing and error-checking purposes
       end = clock();
       elapsed = (end - start) / (double) CLOCKS_PER_SEC;
       printf("GPU: Kernel Execution: %f seconds\n", elapsed);

       // Check for kernel errors
       check_error(cudaGetLastError());

       // Transfer result back to CPU
       start = clock();
       copyResultFromGPU();
       end = clock();
       elapsed = (end - start) / (double) CLOCKS_PER_SEC;
       printf("GPU: Transfer from GPU: %f seconds\n", elapsed);

       // Free the GPU memory
       check_error(cudaFree(A_GPU));
       check_error(cudaFree(B_GPU));
       check_error(cudaFree(C_GPU));
}
```