Machine Problem 2

ELEC 374

Bradley Stephen | 19bbs2 | 20207842

March 20th, 2025

*“I do hereby verify that this machine problem submission is my own work and contains my own original ideas, concepts, and designs. No portion of this report or code has been copied in whole or in part from another source, with the possible exception of properly referenced material"*

# Introduction

In this assignment I implemented a GPU-accelerated tiled matrix multiplication kernel using CUDA shared memory. The objective was to perform multiplication on square matrices with single-precision floating-point elements. The kernel computes one output element per thread by loading sub-blocks (tiles) of the input matrices into shared memory and iterating over these tiles. The GPU result is then compared with a CPU reference implementation, and the correctness is verified if the difference is within a tolerance of 1e-3.

I experimented with different matrix sizes (256×256, 512×512, 1024×1024, 2048×2048, and 4096×4096) and various tile widths (2, 4, 8, 16, and 32). The timings of the GPU kernel execution were measured using CUDA events, repeated 10 times for each configuration to compute average times and standard deviations, while ignoring data transfer and memory allocation times.

# Methodology

I developed the kernel so that each thread calculates a single element of the output matrix P. The kernel uses dynamic shared memory to store tiles from the input matrices M and N, thereby reducing global memory accesses. The matrix is partitioned into sub-blocks (tiles), and each block loads one tile from M and one tile from N into shared memory. Once the tiles are loaded, each thread computes partial products over the tile and accumulates the results to compute its output element. Synchronization is enforced using \_\_syncthreads() between tile loads and computations to ensure that all threads have completed data loading before multiplication begins, and again after processing each tile.

On the host side, for each matrix size, memory is allocated for matrices M, N, and P. The matrices are initialized with random values and transferred to the GPU. The kernel is then launched with a grid and block configuration determined by the chosen tile width. After execution, the output is copied back to the host, and a CPU reference result is computed using a standard triple-nested loop. The GPU result is compared with the CPU reference using a tolerance of 1e-3. GPU kernel execution time is measured using CUDA events over 10 iterations, and both the average time and standard deviation are computed. The CPU timing depends solely on the matrix size, while the GPU timing varies with both matrix size and tile width.

Full code implementation in APPENDIX A

# Results

The following output in table 1 shows the results obtained on a NVIDIA GeForce RTX 3060 Ti. The device has 38 SMs, 4864 CUDA cores in total, and a maximum of 1024 threads per block, with 48 KB of shared memory per block.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| MatrixSize | TileWidth | CPUTimeMs | GPUKernelTimeAvgMs | GPUKernelTimeStdMs | HostToDeviceAvgMs | HostToDeviceStdMs | DeviceToHostAvgMs | DeviceToHostStdMs |
| 256 | 2 | 47.5171 | 15.07504 | 20.00773 | 0.18377 | 0.09357 | 0.11405 | 0.0337 |
| 256 | 4 | 47.5171 | 1.63968 | 0.00685 | 0.12175 | 0.01274 | 0.09978 | 0.01901 |
| 256 | 8 | 47.5171 | 0.41804 | 0.02201 | 0.10949 | 0.01608 | 0.09871 | 0.01169 |
| 256 | 16 | 47.5171 | 0.369 | 0.00224 | 0.10267 | 0.00506 | 0.08944 | 0.00282 |
| 256 | 32 | 47.5171 | 0.31637 | 0.02533 | 0.10055 | 0.00518 | 0.09087 | 0.00656 |
| 512 | 2 | 397.7557 | 56.51291 | 0.73445 | 0.54094 | 0.08042 | 0.34019 | 0.04618 |
| 512 | 4 | 397.7557 | 10.68576 | 0.36239 | 0.50834 | 0.10996 | 0.30909 | 0.06281 |
| 512 | 8 | 397.7557 | 2.34516 | 0.18468 | 0.42072 | 0.04058 | 0.30971 | 0.04372 |
| 512 | 16 | 397.7557 | 1.6654 | 0.21344 | 0.4049 | 0.06508 | 0.3432 | 0.05059 |
| 512 | 32 | 397.7557 | 2.00618 | 0.17895 | 0.35745 | 0.02595 | 0.30691 | 0.03551 |
| 1024 | 2 | 3941.47 | 488.6469 | 9.14208 | 1.79925 | 0.18086 | 0.97134 | 0.03775 |
| 1024 | 4 | 3941.47 | 89.28753 | 0.98016 | 1.93432 | 0.26934 | 0.96184 | 0.06972 |
| 1024 | 8 | 3941.47 | 18.55863 | 0.32527 | 1.773 | 0.20083 | 0.95332 | 0.0964 |
| 1024 | 16 | 3941.47 | 12.30597 | 0.31375 | 1.7273 | 0.12596 | 0.95897 | 0.04777 |
| 1024 | 32 | 3941.47 | 15.2562 | 0.30331 | 1.80758 | 0.1121 | 1.05319 | 0.15125 |
| 2048 | 2 | 39191.74 | 3936.223 | 50.68016 | 9.07275 | 4.32917 | 4.19176 | 1.61012 |
| 2048 | 4 | 39191.74 | 715.8142 | 2.75718 | 7.15811 | 0.45803 | 3.80738 | 0.35178 |
| 2048 | 8 | 39191.74 | 148.0899 | 0.52891 | 6.83989 | 0.77929 | 3.95188 | 0.70587 |
| 2048 | 16 | 39191.74 | 94.54291 | 0.53867 | 6.91685 | 1.17779 | 3.58901 | 0.31016 |
| 2048 | 32 | 39191.74 | 118.6499 | 0.28212 | 6.91169 | 0.7182 | 3.65353 | 0.16155 |
| 4096 | 2 | 320972.1 | 31628.01 | 210.5486 | 39.46722 | 15.89769 | 17.34344 | 1.87356 |
| 4096 | 4 | 320972.1 | 5795.459 | 37.19167 | 32.53102 | 5.55431 | 18.20689 | 5.11932 |
| 4096 | 8 | 320972.1 | 1186.251 | 4.40376 | 25.94269 | 1.42181 | 13.83968 | 0.62691 |
| 4096 | 16 | 320972.1 | 760.1201 | 0.64977 | 25.55402 | 1.16586 | 14.03549 | 0.93397 |
| 4096 | 32 | 320972.1 | 962.2989 | 0.16336 | 26.09432 | 2.73722 | 13.85276 | 0.83822 |

For a 256×256 matrix, the CPU reference time was 47.51710 ms. The GPU kernel times were as follows:

* With a tile width of 2, the kernel averaged 15.07504 ms (Std = 20.00773 ms).
* With a tile width of 4, the kernel averaged 1.63968 ms (Std = 0.00685 ms).
* With a tile width of 8, the kernel averaged 0.41804 ms (Std = 0.02201 ms).
* With a tile width of 16, the kernel averaged 0.36900 ms (Std = 0.00224 ms).
* With a tile width of 32, the kernel averaged 0.31637 ms (Std = 0.02533 ms).

For a 512×512 matrix, the CPU time increased to 397.75570 ms and the GPU kernel times ranged from 56.51291 ms (tile width = 2) to 2.00618 ms (tile width = 32).

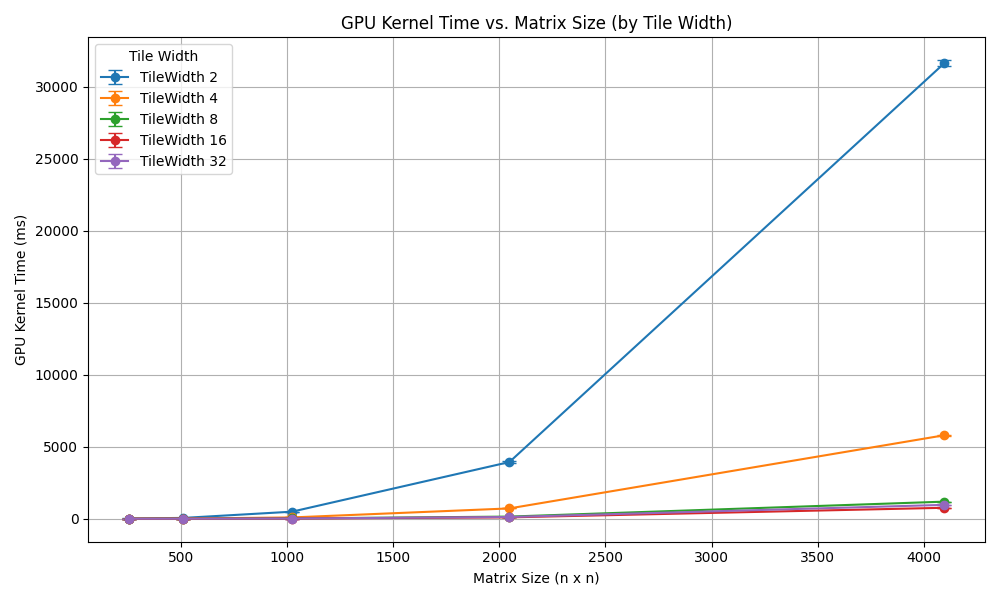
For a 1024×1024 matrix, the CPU time was 3941.46990 ms. The kernel times varied from 488.64686 ms (tile width = 2) down to 15.25620 ms (tile width = 32).

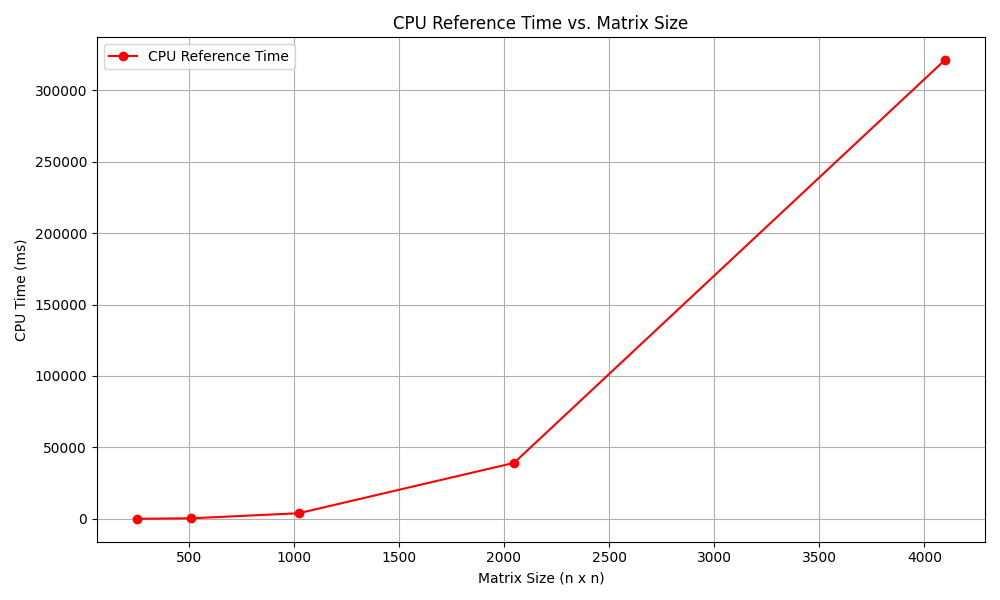
For a 2048×2048 matrix, the CPU reference time was 39191.73690 ms, and the GPU kernel times ranged from 3936.22290 ms (tile width = 2) to 118.64990 ms (tile width = 32).

For a 4096×4096 matrix, the CPU time was 320972.13030 ms, while the GPU kernel times ranged from 31628.01172 ms (tile width = 2) to 962.29892 ms (tile width = 32).

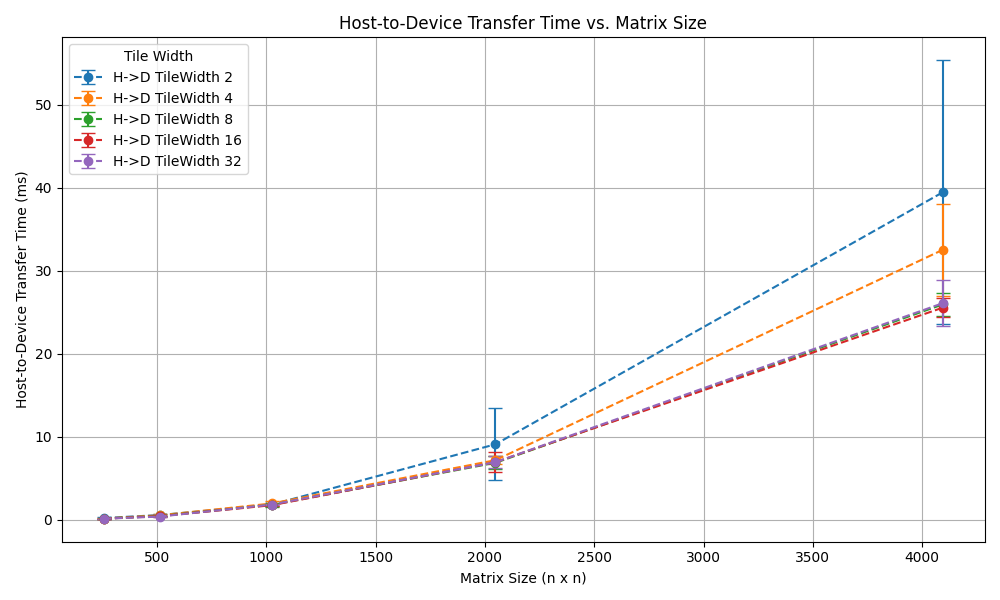
All tests resulted in "PASSED" when the GPU result was compared with the CPU reference.

Below are the four figures generated from the results:

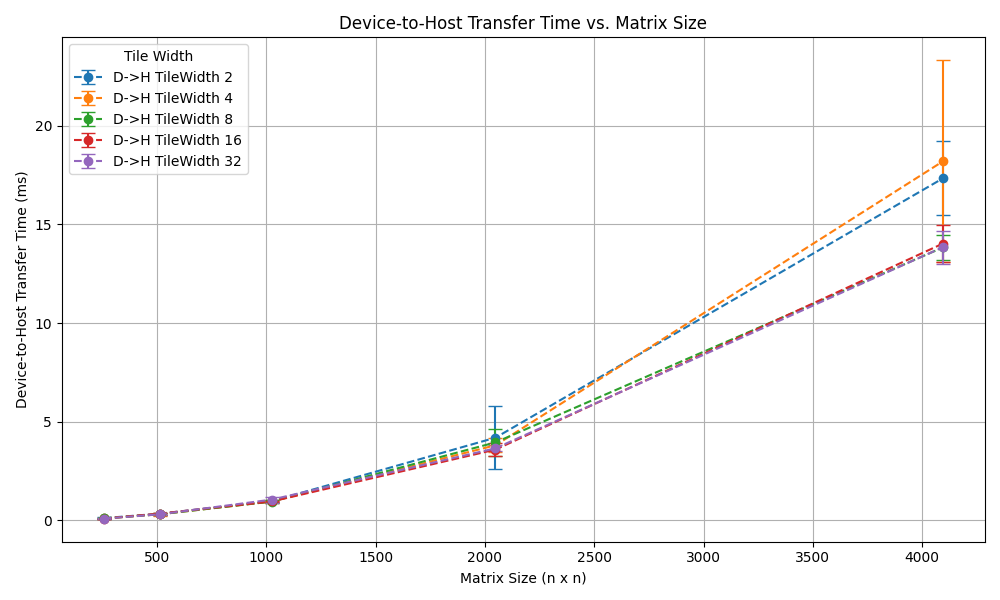




* Figure 3: Host-to-Device Transfer Time vs. Matrix Size for each tile width.



* Figure 4: Device-to-Host Transfer Time vs. Matrix Size for each tile width.



# Discussion

The GPU implementation in Machine Problem 2 clearly outperforms the CPU reference for all tested matrix sizes, especially as the size increases. For instance, at 1024×1024 the CPU requires about 3941 ms, whereas the GPU using a tile width of 16 or 32 finishes in under 16 ms. This performance gap grows even larger for 4096×4096 matrices, where the CPU needs over 320,000 ms and the GPU, with an optimal tile width, can complete the same task in under 1000 ms. These results confirm that the tiled approach with shared memory substantially reduces kernel execution time compared to the baseline CPU implementation.

When comparing to Machine Problem 1 (MP1), which did not fully leverage shared memory for tiling, the new results in MP2 often show faster GPU kernel times at larger tile widths. In MP1, a 256×256 matrix at tile width 32 took about 0.61 ms, whereas MP2 completes the same operation in around 0.316 ms. Similarly, for 1024×1024, MP1 took over 33 ms at tile width 32, whereas MP2 can handle it in about 15 ms. This improvement reflects how the shared memory approach in MP2 reuses data more effectively and thus benefits from fewer global memory accesses, leading to higher throughput.

The CPU reference time remains consistent for a given matrix size, independent of tile width, confirming that only the GPU’s performance depends on that parameter. The figures generated from MP2 data (e.g., Figure 1 for kernel execution time vs. matrix size) highlight the rapid decrease in kernel time as the tile width increases, while Figures illustrating host-to-device and device-to-host transfer times show that data transfers remain relatively small fractions of the overall runtime. This observation further emphasizes that the main performance gains are due to effective tiling in the GPU kernel itself.

In terms of resource usage, the kernel’s dynamic shared memory size is proportional to the square of the tile width. For example, a tile width of 32 requires about 8192 bytes of shared memory per block (2 × 32 × 32 × sizeof(float)). On the NVIDIA GeForce RTX 3060 Ti used in these experiments, there are 38 streaming multiprocessors (SMs), each capable of scheduling up to 2048 threads concurrently (depending on register and shared memory constraints), leading to a theoretical maximum of 77,824 active threads. However, actual occupancy depends on how many blocks can fit on each SM given the shared memory and register requirements of the kernel.

Question 1  
On the RTX 3060 Ti, there are 38 SMs. If each SM can schedule up to 2048 threads, then up to 38 × 2048 = 77,824 threads could theoretically be active at once. In practice, occupancy is limited by shared memory usage and register usage per block.

Question 2  
The kernel uses shared memory sized at 2 × (TILE\_WIDTH × TILE\_WIDTH × sizeof(float)). With TILE\_WIDTH = 32, this is 8 KB per block. Registers per thread can be determined from compilation logs (for example, using nvcc --ptxas-options=-v). Each SM can host multiple blocks if there are sufficient registers and shared memory. Because the device supports up to 48 KB of shared memory per block, it is possible to run several blocks concurrently on one SM, depending on the tile size. The maximum total threads simultaneously scheduled is thus the product of SM count and per-SM concurrency, subject to these resource constraints.

# Part 2: Bonus Mark

In the bonus section, I extended the tiled matrix multiplication kernel to remove two simplifying assumptions: (1) that matrices are square, and (2) that matrix dimensions are exact multiples of the tile dimensions. The revised kernel includes boundary checks when loading tiles from the input matrices and when writing the computed results to the output matrix, ensuring correct operation for non-square matrices with arbitrary dimensions.

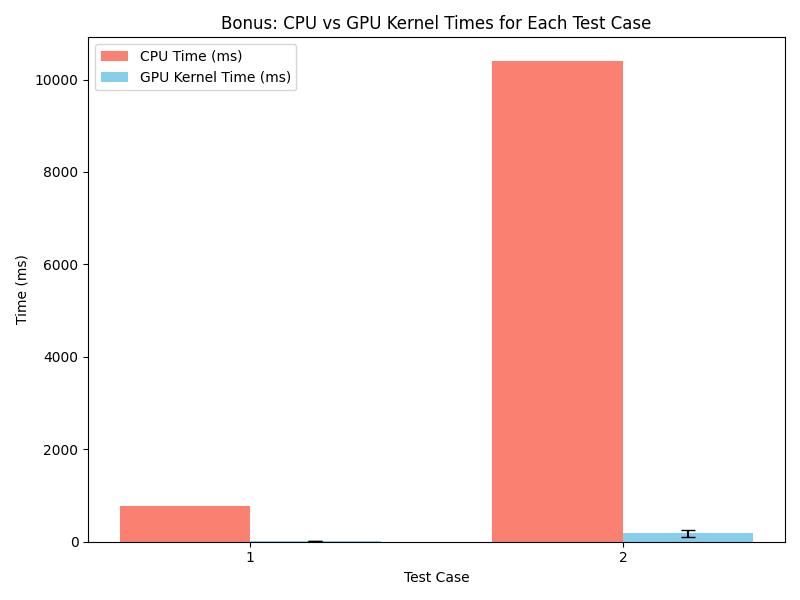
I conducted experiments using two test cases:

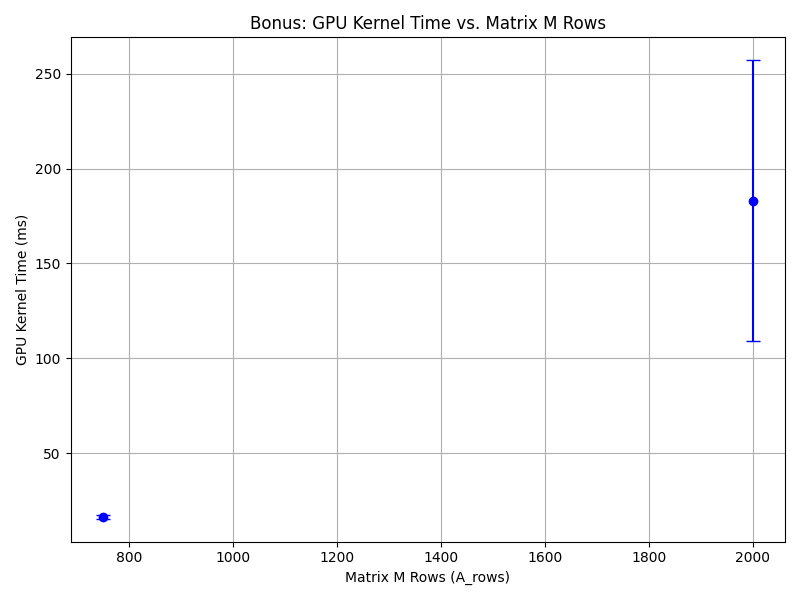
* Test 1: Multiplying a 750×800 matrix M with an 800×850 matrix N, producing a 750×850 output matrix.
* Test 2: Multiplying a 2000×1750 matrix M with a 1750×1900 matrix N, producing a 2000×1900 output matrix.

For these experiments, I used a fixed tile size of 12×18. Kernel execution time was measured using CUDA events over 10 iterations (excluding data transfer and memory allocation times), and the average kernel time and standard deviation were computed. The CPU reference result was computed using a standard triple nested loop, and the GPU output was verified against this reference using a tolerance of 1e-3.

The results indicate that the GPU kernel, even with the additional overhead from boundary checks, still delivers significant speedup compared to the CPU. For example, in Test 1 the CPU took approximately 776.74 ms while the GPU kernel averaged around 16.54 ms (with a standard deviation of about 1.09 ms). In Test 2, the CPU time was around 10393.25 ms, and the GPU kernel averaged 183.02 ms (Std ≈ 74.12 ms). Both test cases passed the correctness check.

Figure 2 (bonus\_cpu\_vs\_gpu.png) presents a bar plot comparing CPU and GPU kernel times for each test case, while Figure 3 (bonus\_gpu\_kernel\_vs\_rows.png) shows a scatter plot of GPU kernel time as a function of the number of rows in matrix M. These figures demonstrate that the revised kernel is robust and efficient even when dealing with matrices of arbitrary dimensions.





# Conclusion

uccessfully implemented a tiled matrix multiplication kernel using CUDA shared memory. The experimental results demonstrate significant speedup over the CPU reference implementation, particularly as matrix size increases and when using optimal tile widths. The kernel scales well with increasing matrix size, and the shared memory optimization significantly reduces global memory accesses, leading to improved throughput. The four figures generated from the results clearly illustrate these performance trends. Moreover, when compared to the earlier approach in MP1, the current implementation (MP2) achieves lower kernel execution times at larger tile widths, confirming the advantages of effective shared memory utilization. The resource usage analysis further validates the efficiency of the tiled approach on the NVIDIA GeForce RTX 3060 Ti, confirming that the design leverages the available CUDA cores and shared memory to deliver high performance.

Appendix A on next page >>>

# APPENDIX A

MP2\_Part1.cu full code:

|  |
| --- |
| *// Bradley Stephen | 19bbs2 | 20207842*  *// March 27th, 2025*  *// ELEC 374 - Machine Problem 2 - Part 1*  *#include <cstdio>*  *#include <cstdlib>*  *#include <cmath>*  *#include <cuda.h>*  *#include <vector>*  *#include <string>*  *#include <chrono>*  *#include <iostream>*  *#define NUM\_ITER 10*  *#define TOLERANCE 1e-3f*  *// Tiled Matrix Multiplication Kernel*  *\_\_global\_\_*  *void tiledMatMulKernel(const float\* \_\_restrict\_\_ M,*  *const float\* \_\_restrict\_\_ N,*  *float\* \_\_restrict\_\_ P,*  *int width,*  *int TILE\_WIDTH)*  *{*  *extern \_\_shared\_\_ float sharedMem[];*  *float\* Mds = sharedMem;*  *float\* Nds = sharedMem + TILE\_WIDTH \* TILE\_WIDTH;*  *int bx = blockIdx.x;*  *int by = blockIdx.y;*  *int tx = threadIdx.x;*  *int ty = threadIdx.y;*  *int Row = by \* TILE\_WIDTH + ty;*  *int Col = bx \* TILE\_WIDTH + tx;*  *float Pvalue = 0.0f;*  *int numTiles = width / TILE\_WIDTH;  // assumes width % TILE\_WIDTH == 0*  *for (int t = 0; t < numTiles; t++)*  *{*  *Mds[ty \* TILE\_WIDTH + tx] = M[Row \* width + (t \* TILE\_WIDTH + tx)];*  *Nds[ty \* TILE\_WIDTH + tx] = N[(t \* TILE\_WIDTH + ty) \* width + Col];*  *\_\_syncthreads();*  *for (int k = 0; k < TILE\_WIDTH; k++)*  *{*  *Pvalue += Mds[ty \* TILE\_WIDTH + k] \* Nds[k \* TILE\_WIDTH + tx];*  *}*  *\_\_syncthreads();*  *}*  *P[Row \* width + Col] = Pvalue;*  *}*  *// CPU Reference Multiplication (for verification)*  *void cpuMatrixMul(const float\* M, const float\* N, float\* P, int width)*  *{*  *for (int i = 0; i < width; i++)*  *{*  *for (int j = 0; j < width; j++)*  *{*  *float sum = 0.0f;*  *for (int k = 0; k < width; k++)*  *{*  *sum += M[i \* width + k] \* N[k \* width + j];*  *}*  *P[i \* width + j] = sum;*  *}*  *}*  *}*  *// Compare GPU result with CPU result*  *bool compareResults(const float\* ref, const float\* gpu, int size, float tolerance = TOLERANCE)*  *{*  *for (int i = 0; i < size; i++)*  *{*  *float diff = fabs(ref[i] - gpu[i]);*  *if (diff > tolerance)*  *{*  *return false;*  *}*  *}*  *return true;*  *}*  *//device query*  *// Query and prinrt CUDA device porps*  *void queryDeviceProperties() {*  *int deviceCount;*  *cudaGetDeviceCount(&deviceCount);*  *printf("CUDA Devices Found: %d\n", deviceCount);*  *if (deviceCount == 0) {*  *printf("No CUDA devices available.\n");*  *return;*  *}*  *for (int i = 0; i < deviceCount; i++) {*  *cudaDeviceProp deviceProp;*  *cudaGetDeviceProperties(&deviceProp, i);*  *// Basic device info*  *printf("\nDevice %d: %s\n", i, deviceProp.name);*  *printf("Compute Capability: %d.%d\n", deviceProp.major, deviceProp.minor);*  *printf("Clock Speed: %d kHz\n", deviceProp.clockRate);*  *printf("Streaming Multiprocessors (SMs): %d\n", deviceProp.multiProcessorCount);*  *// Determine CUDA core count based on specifc arc.*  *int coresPerSM;*  *switch (deviceProp.major) {*  *case 2:  coresPerSM = (deviceProp.minor == 1) ? 48 : 32; break;  // Fermi*  *case 3:  coresPerSM = 192; break;  // Kepler*  *case 5:  coresPerSM = 128; break;  // Maxwell*  *case 6:  coresPerSM = (deviceProp.minor == 1) ? 128 : 64; break;  // Pascal*  *case 7:  coresPerSM = 64; break;  // Volta/Turing*  *case 8:  coresPerSM = (deviceProp.minor == 6 || deviceProp.minor == 9) ? 128 : 64; break;  // Ampere*  *default: coresPerSM = 64;  // Default: 64*  *}*  *// Print Results*  *int totalCores = coresPerSM \* deviceProp.multiProcessorCount;*  *printf("CUDA Cores per SM: %d\n", coresPerSM);*  *printf("Total CUDA Cores: %d\n", totalCores);*  *printf("Warp Size: %d\n", deviceProp.warpSize);*  *// Mem sizes*  *printf("Global Memory: %.2f GB\n", deviceProp.totalGlobalMem / (1024.0 \* 1024.0 \* 1024.0));*  *printf("Constant Memory: %.2f KB\n", deviceProp.totalConstMem / 1024.0);*  *printf("Shared Memory per Block: %.2f KB\n", deviceProp.sharedMemPerBlock / 1024.0);*  *printf("Registers per Block: %d\n", deviceProp.regsPerBlock);*  *printf("Max Threads per Block: %d\n", deviceProp.maxThreadsPerBlock);*  *// Max limits for parallel EX*  *printf("Max Block Dimensions: (%d, %d, %d)\n",*  *deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);*  *printf("Max Grid Dimensions: (%d, %d, %d)\n",*  *deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);*  *}*  *}*  *// Main*  *int main()*  *{*  *// For easy refernce*  *queryDeviceProperties();*  *printf("============================================================\n");*  *printf("   Machine Problem 2: Tiled Matrix Multiplication (GPU)   \n");*  *printf("============================================================\n\n");*  *std::vector<int> matrixSizes = { 256, 512, 1024, 2048, 4096 };*  *std::vector<int> tileWidths = { 2, 4, 8, 16, 32 };*  *// Open CSV file for results with extended columns for standard deviations*  *FILE\* fp = fopen("MP2.csv", "w");*  *if (!fp)*  *{*  *fprintf(stderr, "Error opening MP2.csv for writing.\n");*  *return -1;*  *}*  *fprintf(fp, "MatrixSize,TileWidth,CPUTimeMs,GPUKernelTimeAvgMs,GPUKernelTimeStdMs,HostToDeviceAvgMs,HostToDeviceStdMs,DeviceToHostAvgMs,DeviceToHostStdMs\n");*  *// Loop over each matrix size*  *for (int size : matrixSizes)*  *{*  *size\_t bytes = static\_cast<size\_t>(size) \* size \* sizeof(float);*  *float\* h\_M = (float\*)malloc(bytes);*  *float\* h\_N = (float\*)malloc(bytes);*  *float\* h\_P = (float\*)malloc(bytes);   // GPU result*  *float\* h\_ref = (float\*)malloc(bytes); // CPU reference*  *srand(0); // fixed seed for reproducibility*  *for (int i = 0; i < size \* size; i++)*  *{*  *h\_M[i] = static\_cast<float>(rand() % 100) / 10.0f;*  *h\_N[i] = static\_cast<float>(rand() % 100) / 10.0f;*  *}*  *// Compute CPU reference and measure CPU time*  *auto cpuStart = std::chrono::high\_resolution\_clock::now();*  *cpuMatrixMul(h\_M, h\_N, h\_ref, size);*  *auto cpuEnd = std::chrono::high\_resolution\_clock::now();*  *double cpuMs = std::chrono::duration<double, std::milli>(cpuEnd - cpuStart).count();*  *printf("------------------------------------------------------------\n");*  *printf("Matrix Size: %d x %d\n", size, size);*  *printf("CPU Reference Time: %.5f ms\n", cpuMs);*  *printf("------------------------------------------------------------\n");*  *// Loop over each tile width*  *for (int TW : tileWidths)*  *{*  *if (size % TW != 0)*  *{*  *printf("  [TileWidth=%d] Skipped (matrix not divisible by tile width)\n", TW);*  *continue;*  *}*  *// Allocate device memory once for this configuration*  *float\* d\_M, \* d\_N, \* d\_P;*  *cudaMalloc((void\*\*)&d\_M, bytes);*  *cudaMalloc((void\*\*)&d\_N, bytes);*  *cudaMalloc((void\*\*)&d\_P, bytes);*  *// Setup kernel launch parameters*  *dim3 dimBlock(TW, TW);*  *dim3 dimGrid(size / TW, size / TW);*  *size\_t sharedMemSize = 2 \* TW \* TW \* sizeof(float);*  *double sumKernel = 0.0, sumH2d = 0.0, sumD2h = 0.0;*  *double sumSqKernel = 0.0, sumSqH2d = 0.0, sumSqD2h = 0.0;*  *// Run NUM\_ITER iterations for timing*  *for (int iter = 0; iter < NUM\_ITER; iter++)*  *{*  *// Host-to-Device transfer timing*  *cudaEvent\_t h2dStart, h2dStop;*  *cudaEventCreate(&h2dStart);*  *cudaEventCreate(&h2dStop);*  *cudaEventRecord(h2dStart);*  *cudaMemcpy(d\_M, h\_M, bytes, cudaMemcpyHostToDevice);*  *cudaMemcpy(d\_N, h\_N, bytes, cudaMemcpyHostToDevice);*  *cudaEventRecord(h2dStop);*  *cudaEventSynchronize(h2dStop);*  *float h2dMs = 0.0f;*  *cudaEventElapsedTime(&h2dMs, h2dStart, h2dStop);*  *cudaEventDestroy(h2dStart);*  *cudaEventDestroy(h2dStop);*  *// Kernel execution timing*  *cudaEvent\_t kStart, kStop;*  *cudaEventCreate(&kStart);*  *cudaEventCreate(&kStop);*  *cudaEventRecord(kStart);*  *tiledMatMulKernel << <dimGrid, dimBlock, sharedMemSize >> > (d\_M, d\_N, d\_P, size, TW);*  *cudaEventRecord(kStop);*  *cudaEventSynchronize(kStop);*  *float kernelMs = 0.0f;*  *cudaEventElapsedTime(&kernelMs, kStart, kStop);*  *cudaEventDestroy(kStart);*  *cudaEventDestroy(kStop);*  *// Device-to-Host transfer timing*  *cudaEvent\_t d2hStart, d2hStop;*  *cudaEventCreate(&d2hStart);*  *cudaEventCreate(&d2hStop);*  *cudaEventRecord(d2hStart);*  *cudaMemcpy(h\_P, d\_P, bytes, cudaMemcpyDeviceToHost);*  *cudaEventRecord(d2hStop);*  *cudaEventSynchronize(d2hStop);*  *float d2hMs = 0.0f;*  *cudaEventElapsedTime(&d2hMs, d2hStart, d2hStop);*  *cudaEventDestroy(d2hStart);*  *cudaEventDestroy(d2hStop);*  *sumH2d += h2dMs;*  *sumKernel += kernelMs;*  *sumD2h += d2hMs;*  *sumSqH2d += h2dMs \* h2dMs;*  *sumSqKernel += kernelMs \* kernelMs;*  *sumSqD2h += d2hMs \* d2hMs;*  *}*  *// Compute averages*  *double avgH2d = sumH2d / NUM\_ITER;*  *double avgKernel = sumKernel / NUM\_ITER;*  *double avgD2h = sumD2h / NUM\_ITER;*  *// Compute standard deviations*  *double stdH2d = sqrt((sumSqH2d / NUM\_ITER) - (avgH2d \* avgH2d));*  *double stdKernel = sqrt((sumSqKernel / NUM\_ITER) - (avgKernel \* avgKernel));*  *double stdD2h = sqrt((sumSqD2h / NUM\_ITER) - (avgD2h \* avgD2h));*  *// Verify correctness using result from last iteration*  *bool pass = compareResults(h\_ref, h\_P, size \* size);*  *printf("  [TileWidth=%2d] Kernel: Avg = %.5f ms (Std = %.5f ms) | H->D: Avg = %.5f ms (Std = %.5f ms) | D->H: Avg = %.5f ms (Std = %.5f ms) | ",*  *TW, avgKernel, stdKernel, avgH2d, stdH2d, avgD2h, stdD2h);*  *if (pass) printf("Result: PASSED\n");*  *else     printf("Result: FAILED\n");*  *// Write averaged results to CSV*  *fprintf(fp, "%d,%d,%.5f,%.5f,%.5f,%.5f,%.5f,%.5f,%.5f\n",*  *size, TW, cpuMs, avgKernel, stdKernel, avgH2d, stdH2d, avgD2h, stdD2h);*  *// Cleanup device memory for this configuration*  *cudaFree(d\_M);*  *cudaFree(d\_N);*  *cudaFree(d\_P);*  *}*  *free(h\_M);*  *free(h\_N);*  *free(h\_P);*  *free(h\_ref);*  *printf("\n");*  *}*  *fclose(fp);*  *printf("============================================================\n");*  *printf("All results written to MP2.csv\n");*  *printf("============================================================\n");*  *return 0;*  *}* |

Part 1 terminal output screenshot:

A screen shot of a computer

AI-generated content may be incorrect.

Part 1 terminal output raw:

|  |
| --- |
| CUDA Devices Found: 1  Device 0: NVIDIA GeForce RTX 3060 Ti  Compute Capability: 8.6  Clock Speed: 1665000 kHz  Streaming Multiprocessors (SMs): 38  CUDA Cores per SM: 128  Total CUDA Cores: 4864  Warp Size: 32  Global Memory: 8.00 GB  Constant Memory: 64.00 KB  Shared Memory per Block: 48.00 KB  Registers per Block: 65536  Max Threads per Block: 1024  Max Block Dimensions: (1024, 1024, 64)  Max Grid Dimensions: (2147483647, 65535, 65535)  ============================================================  Machine Problem 2: Tiled Matrix Multiplication (GPU)  ============================================================  ------------------------------------------------------------  Matrix Size: 256 x 256  CPU Reference Time: 47.51710 ms  ------------------------------------------------------------  [TileWidth= 2] Kernel: Avg = 15.07504 ms (Std = 20.00773 ms) | H->D: Avg = 0.18377 ms (Std = 0.09357 ms) | D->H: Avg = 0.11405 ms (Std = 0.03370 ms) | Result: PASSED  [TileWidth= 4] Kernel: Avg = 1.63968 ms (Std = 0.00685 ms) | H->D: Avg = 0.12175 ms (Std = 0.01274 ms) | D->H: Avg = 0.09978 ms (Std = 0.01901 ms) | Result: PASSED  [TileWidth= 8] Kernel: Avg = 0.41804 ms (Std = 0.02201 ms) | H->D: Avg = 0.10949 ms (Std = 0.01608 ms) | D->H: Avg = 0.09871 ms (Std = 0.01169 ms) | Result: PASSED  [TileWidth=16] Kernel: Avg = 0.36900 ms (Std = 0.00224 ms) | H->D: Avg = 0.10267 ms (Std = 0.00506 ms) | D->H: Avg = 0.08944 ms (Std = 0.00282 ms) | Result: PASSED  [TileWidth=32] Kernel: Avg = 0.31637 ms (Std = 0.02533 ms) | H->D: Avg = 0.10055 ms (Std = 0.00518 ms) | D->H: Avg = 0.09087 ms (Std = 0.00656 ms) | Result: PASSED  ------------------------------------------------------------  Matrix Size: 512 x 512  CPU Reference Time: 397.75570 ms  ------------------------------------------------------------  [TileWidth= 2] Kernel: Avg = 56.51291 ms (Std = 0.73445 ms) | H->D: Avg = 0.54094 ms (Std = 0.08042 ms) | D->H: Avg = 0.34019 ms (Std = 0.04618 ms) | Result: PASSED  [TileWidth= 4] Kernel: Avg = 10.68576 ms (Std = 0.36239 ms) | H->D: Avg = 0.50834 ms (Std = 0.10996 ms) | D->H: Avg = 0.30909 ms (Std = 0.06281 ms) | Result: PASSED  [TileWidth= 8] Kernel: Avg = 2.34516 ms (Std = 0.18468 ms) | H->D: Avg = 0.42072 ms (Std = 0.04058 ms) | D->H: Avg = 0.30971 ms (Std = 0.04372 ms) | Result: PASSED  [TileWidth=16] Kernel: Avg = 1.66540 ms (Std = 0.21344 ms) | H->D: Avg = 0.40490 ms (Std = 0.06508 ms) | D->H: Avg = 0.34320 ms (Std = 0.05059 ms) | Result: PASSED  [TileWidth=32] Kernel: Avg = 2.00618 ms (Std = 0.17895 ms) | H->D: Avg = 0.35745 ms (Std = 0.02595 ms) | D->H: Avg = 0.30691 ms (Std = 0.03551 ms) | Result: PASSED  ------------------------------------------------------------  Matrix Size: 1024 x 1024  CPU Reference Time: 3941.46990 ms  ------------------------------------------------------------  [TileWidth= 2] Kernel: Avg = 488.64686 ms (Std = 9.14208 ms) | H->D: Avg = 1.79925 ms (Std = 0.18086 ms) | D->H: Avg = 0.97134 ms (Std = 0.03775 ms) | Result: PASSED  [TileWidth= 4] Kernel: Avg = 89.28753 ms (Std = 0.98016 ms) | H->D: Avg = 1.93432 ms (Std = 0.26934 ms) | D->H: Avg = 0.96184 ms (Std = 0.06972 ms) | Result: PASSED  [TileWidth= 8] Kernel: Avg = 18.55863 ms (Std = 0.32527 ms) | H->D: Avg = 1.77300 ms (Std = 0.20083 ms) | D->H: Avg = 0.95332 ms (Std = 0.09640 ms) | Result: PASSED  [TileWidth=16] Kernel: Avg = 12.30597 ms (Std = 0.31375 ms) | H->D: Avg = 1.72730 ms (Std = 0.12596 ms) | D->H: Avg = 0.95897 ms (Std = 0.04777 ms) | Result: PASSED  [TileWidth=32] Kernel: Avg = 15.25620 ms (Std = 0.30331 ms) | H->D: Avg = 1.80758 ms (Std = 0.11210 ms) | D->H: Avg = 1.05319 ms (Std = 0.15125 ms) | Result: PASSED  ------------------------------------------------------------  Matrix Size: 2048 x 2048  CPU Reference Time: 39191.73690 ms  ------------------------------------------------------------  [TileWidth= 2] Kernel: Avg = 3936.22290 ms (Std = 50.68016 ms) | H->D: Avg = 9.07275 ms (Std = 4.32917 ms) | D->H: Avg = 4.19176 ms (Std = 1.61012 ms) | Result: PASSED  [TileWidth= 4] Kernel: Avg = 715.81423 ms (Std = 2.75718 ms) | H->D: Avg = 7.15811 ms (Std = 0.45803 ms) | D->H: Avg = 3.80738 ms (Std = 0.35178 ms) | Result: PASSED  [TileWidth= 8] Kernel: Avg = 148.08987 ms (Std = 0.52891 ms) | H->D: Avg = 6.83989 ms (Std = 0.77929 ms) | D->H: Avg = 3.95188 ms (Std = 0.70587 ms) | Result: PASSED  [TileWidth=16] Kernel: Avg = 94.54291 ms (Std = 0.53867 ms) | H->D: Avg = 6.91685 ms (Std = 1.17779 ms) | D->H: Avg = 3.58901 ms (Std = 0.31016 ms) | Result: PASSED  [TileWidth=32] Kernel: Avg = 118.64990 ms (Std = 0.28212 ms) | H->D: Avg = 6.91169 ms (Std = 0.71820 ms) | D->H: Avg = 3.65353 ms (Std = 0.16155 ms) | Result: PASSED  ------------------------------------------------------------  Matrix Size: 4096 x 4096  CPU Reference Time: 320972.13030 ms  ------------------------------------------------------------  [TileWidth= 2] Kernel: Avg = 31628.01172 ms (Std = 210.54861 ms) | H->D: Avg = 39.46722 ms (Std = 15.89769 ms) | D->H: Avg = 17.34344 ms (Std = 1.87356 ms) | Result: PASSED  [TileWidth= 4] Kernel: Avg = 5795.45850 ms (Std = 37.19167 ms) | H->D: Avg = 32.53102 ms (Std = 5.55431 ms) | D->H: Avg = 18.20689 ms (Std = 5.11932 ms) | Result: PASSED  [TileWidth= 8] Kernel: Avg = 1186.25083 ms (Std = 4.40376 ms) | H->D: Avg = 25.94269 ms (Std = 1.42181 ms) | D->H: Avg = 13.83968 ms (Std = 0.62691 ms) | Result: PASSED  [TileWidth=16] Kernel: Avg = 760.12006 ms (Std = 0.64977 ms) | H->D: Avg = 25.55402 ms (Std = 1.16586 ms) | D->H: Avg = 14.03549 ms (Std = 0.93397 ms) | Result: PASSED  [TileWidth=32] Kernel: Avg = 962.29892 ms (Std = 0.16336 ms) | H->D: Avg = 26.09432 ms (Std = 2.73722 ms) | D->H: Avg = 13.85276 ms (Std = 0.83822 ms) | Result: PASSED  ============================================================  All results written to MP2.csv  ============================================================  C:\Users\19bbs2\source\repos\MP2\_ELEC374\_Bradley\x64\Debug\MP2\_ELEC374\_Bradley.exe (process 14944) exited with code 0.  Press any key to close this window . . . |

MP2 Part 2 Bonus full code:

|  |
| --- |
| // Bradley Stephen | 19bbs2 | 20207842  // March 27th, 2025  // ELEC 374 - Machine Problem 2 - Part 2 bonus  #include <cstdio>  #include <cstdlib>  #include <cmath>  #include <cuda.h>  #include <chrono>  #define NUM\_ITER 10  #define TOLERANCE 1e-3f  #define TILE\_ROW 12  #define TILE\_COL 18  #define TILE\_COMMON 18  // Revised tiled matrix multiplication kernel with boundary checks.  // Computes: P = M \* N, where M is A\_rows x A\_cols and N is A\_cols x B\_cols.  \_\_global\_\_  *void* tiledMatMulKernelBoundary(const *float*\* \_\_restrict\_\_ *M*,      const *float*\* \_\_restrict\_\_ *N*,  *float*\* \_\_restrict\_\_ *P*,  *int* *A\_rows*, *int* *A\_cols*, *int* *B\_cols*)  {      extern \_\_shared\_\_ *float* shared[];  *float*\* tileM = shared;                      // TILE\_ROW x TILE\_COMMON  *float*\* tileN = shared + TILE\_ROW \* TILE\_COMMON;  // TILE\_COMMON x TILE\_COL  *int* row = *blockIdx*.y \* TILE\_ROW + *threadIdx*.y;  *int* col = *blockIdx*.x \* TILE\_COL + *threadIdx*.x;  *float* Pvalue = 0.0f;  *int* numTiles = (A\_cols + TILE\_COMMON - 1) / TILE\_COMMON;      for (*int* t = 0; t < numTiles; t++)      {  *int* m\_col = t \* TILE\_COMMON + *threadIdx*.x;          if (row < A\_rows && m\_col < A\_cols)  *tileM*[*threadIdx*.*y* \* TILE\_COMMON + *threadIdx*.x] = *M*[row \* A\_cols + m\_col];          else  *tileM*[*threadIdx*.*y* \* TILE\_COMMON + *threadIdx*.x] = 0.0f;          for (*int* i = *threadIdx*.y; i < TILE\_COMMON; i += TILE\_ROW)          {  *int* n\_row = t \* TILE\_COMMON + i;              if (n\_row < A\_cols && col < B\_cols)  *tileN*[i \* TILE\_COL + *threadIdx*.x] = *N*[n\_row \* B\_cols + col];              else  *tileN*[i \* TILE\_COL + *threadIdx*.x] = 0.0f;          }          \_\_syncthreads();          for (*int* k = 0; k < TILE\_COMMON; k++)              Pvalue += *tileM*[*threadIdx*.*y* \* TILE\_COMMON + k] \* *tileN*[k \* TILE\_COL + *threadIdx*.x];          \_\_syncthreads();      }      if (row < A\_rows && col < B\_cols)  *P*[row \* B\_cols + col] = Pvalue;  }  *void* cpuMatrixMulBoundary(const *float*\* *M*, const *float*\* *N*, *float*\* *P*,  *int* *A\_rows*, *int* *A\_cols*, *int* *B\_cols*)  {      for (*int* i = 0; i < A\_rows; i++)      {          for (*int* j = 0; j < B\_cols; j++)          {  *float* sum = 0.0f;              for (*int* k = 0; k < A\_cols; k++)                  sum += *M*[i \* A\_cols + k] \* *N*[k \* B\_cols + j];  *P*[i \* B\_cols + j] = sum;          }      }  }  *bool* compareResultsBoundary(const *float*\* *ref*, const *float*\* *gpu*, *int* *size*, *float* *tolerance* = TOLERANCE)  {      for (*int* i = 0; i < size; i++)      {          if (fabs(*ref*[i] - *gpu*[i]) > tolerance)              return false;      }      return true;  }  *int* main()  {      FILE\* fp = fopen("MP2Bonus.csv", "w");      if (!fp)      {          printf("Error opening MP2Bonus.csv for writing.\n");          return -1;      }      // CSV header: Test,A\_rows,A\_cols,B\_cols,CPUTimeMs,GPUKernelAvgMs,GPUKernelStdMs,Result      fprintf(fp, "Test,A\_rows,A\_cols,B\_cols,CPUTimeMs,GPUKernelAvgMs,GPUKernelStdMs,Result\n");      // Test cases:      // Test 1: M: 750 x 800, N: 800 x 850 => P: 750 x 850      // Test 2: M: 2000 x 1750, N: 1750 x 1900 => P: 2000 x 1900  *struct* TestCase {  *int* A\_rows, A\_cols, B\_cols;      } *tests*[2] = { {750, 800, 850}, {2000, 1750, 1900} };      for (*int* test = 0; test < 2; test++)      {  *int* A\_rows = *tests*[test].*A\_rows*;  *int* A\_cols = *tests*[test].*A\_cols*;  *int* B\_cols = *tests*[test].*B\_cols*;  *size\_t* sizeM = A\_rows \* A\_cols \* sizeof(*float*);  *size\_t* sizeN = A\_cols \* B\_cols \* sizeof(*float*);  *size\_t* sizeP = A\_rows \* B\_cols \* sizeof(*float*);          printf("Test %d: M: %d x %d, N: %d x %d, P: %d x %d\n", test + 1, A\_rows, A\_cols, A\_cols, B\_cols, A\_rows, B\_cols);  *float*\* h\_M = (*float*\*)malloc(sizeM);  *float*\* h\_N = (*float*\*)malloc(sizeN);  *float*\* h\_P = (*float*\*)malloc(sizeP);  *float*\* h\_ref = (*float*\*)malloc(sizeP);          srand(0);          for (*int* i = 0; i < A\_rows \* A\_cols; i++)  *h\_M*[i] = static\_cast<*float*>(rand() % 100) / 10.0f;          for (*int* i = 0; i < A\_cols \* B\_cols; i++)  *h\_N*[i] = static\_cast<*float*>(rand() % 100) / 10.0f;          // CPU reference multiplication timing  *auto* cpuStart = std::chrono::high\_resolution\_clock::now();          cpuMatrixMulBoundary(h\_M, h\_N, h\_ref, A\_rows, A\_cols, B\_cols);  *auto* cpuEnd = std::chrono::high\_resolution\_clock::now();  *double* cpuTime = std::chrono::duration<*double*, std::milli>(cpuEnd - cpuStart).count();          printf("Test %d: CPU Reference Time: %.5f ms\n", test + 1, cpuTime);  *float*\* d\_M, \* d\_N, \* d\_P;          cudaMalloc((*void*\*\*)&d\_M, sizeM);          cudaMalloc((*void*\*\*)&d\_N, sizeN);          cudaMalloc((*void*\*\*)&d\_P, sizeP);          cudaMemcpy(d\_M, h\_M, sizeM, cudaMemcpyHostToDevice);          cudaMemcpy(d\_N, h\_N, sizeN, cudaMemcpyHostToDevice);  *dim3* dimBlock(TILE\_COL, TILE\_ROW);  // 18 x 12 threads per block  *dim3* dimGrid((B\_cols + TILE\_COL - 1) / TILE\_COL, (A\_rows + TILE\_ROW - 1) / TILE\_ROW);  *size\_t* sharedMemSize = (TILE\_ROW \* TILE\_COMMON + TILE\_COMMON \* TILE\_COL) \* sizeof(*float*);  *double* sumKernel = 0.0, sumSqKernel = 0.0;          for (*int* iter = 0; iter < NUM\_ITER; iter++)          {  *cudaEvent\_t* start, stop;              cudaEventCreate(&start);              cudaEventCreate(&stop);              cudaEventRecord(start);              tiledMatMulKernelBoundary << <dimGrid, dimBlock, sharedMemSize >> > (d\_M, d\_N, d\_P, A\_rows, A\_cols, B\_cols);              cudaEventRecord(stop);              cudaEventSynchronize(stop);  *float* kernelTime = 0.0f;              cudaEventElapsedTime(&kernelTime, start, stop);              cudaEventDestroy(start);              cudaEventDestroy(stop);              sumKernel += kernelTime;              sumSqKernel += kernelTime \* kernelTime;          }  *double* avgKernel = sumKernel / NUM\_ITER;  *double* stdKernel = sqrt((sumSqKernel / NUM\_ITER) - (avgKernel \* avgKernel));          printf("Test %d: GPU Kernel Time: Avg = %.5f ms, Std = %.5f ms\n", test + 1, avgKernel, stdKernel);          cudaMemcpy(h\_P, d\_P, sizeP, cudaMemcpyDeviceToHost);  *bool* correct = compareResultsBoundary(h\_ref, h\_P, A\_rows \* B\_cols);          printf("Test %d: Result %s\n", test + 1, correct ? "PASSED" : "FAILED");          printf("------------------------------------------------------------\n");          fprintf(fp, "%d,%d,%d,%d,%.5f,%.5f,%.5f,%s\n",              test + 1, A\_rows, A\_cols, B\_cols, cpuTime, avgKernel, stdKernel, correct ? "PASSED" : "FAILED");          cudaFree(d\_M);          cudaFree(d\_N);          cudaFree(d\_P);          free(h\_M);          free(h\_N);          free(h\_P);          free(h\_ref);      }      fclose(fp);      printf("Results written to MP2Bonus.csv\n");      return 0;  } |

Part 2 bonus terminal output screenshot:

A computer screen shot of a black screen

AI-generated content may be incorrect.

Part 2 bonus raw terminal output:

|  |
| --- |
| Test 1: M: 750 x 800, N: 800 x 850, P: 750 x 850  Test 1: CPU Reference Time: 1453.03080 ms  Test 1: GPU Kernel Time: Avg = 10.78301 ms, Std = 6.02688 ms  Test 1: Result PASSED  ------------------------------------------------------------  Test 2: M: 2000 x 1750, N: 1750 x 1900, P: 2000 x 1900  Test 2: CPU Reference Time: 28980.55750 ms  Test 2: GPU Kernel Time: Avg = 114.36705 ms, Std = 57.98438 ms  Test 2: Result PASSED  ------------------------------------------------------------  Results written to MP2Bonus.csv  C:\Users\19bbs2\source\repos\MP2\_ELEC374\_Bradley\x64\Debug\MP2\_ELEC374\_Bradley.exe (process 21296) exited with code 0.  Press any key to close this window . . . |