ELEC 374 – Machine Problem 1

"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"

Part 1

Outputs.txt

```
Part 1:
Device 0: NVIDIA T600
Compute Capability: 7.5
Clock Rate: 1335.00 MHz
Number of Streaming Multiprocessors (SMs): 10
Total CUDA Cores (approx.): 640
Warp Size: 32
Global Memory: 4.00 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)
```

Figure 1: Output data of Part 1 script showing All GPUs and their specifications on the machine

The values above were obtained by running the MP1_Part1.cu code. Some of values above were verified using the data sheet at the following link (leads to a pdf). To obtain the total number of CUDA cores the major and minor number were obtained using the attributes in the deviceProp. That had to be cross-referenced with information obtained online to determine the number of CUDA cores for any given GPU. The program cannot determine the CUDA cores for the new Blackwell architecture as the reference I was using was not up to date.

Part 2

2.1

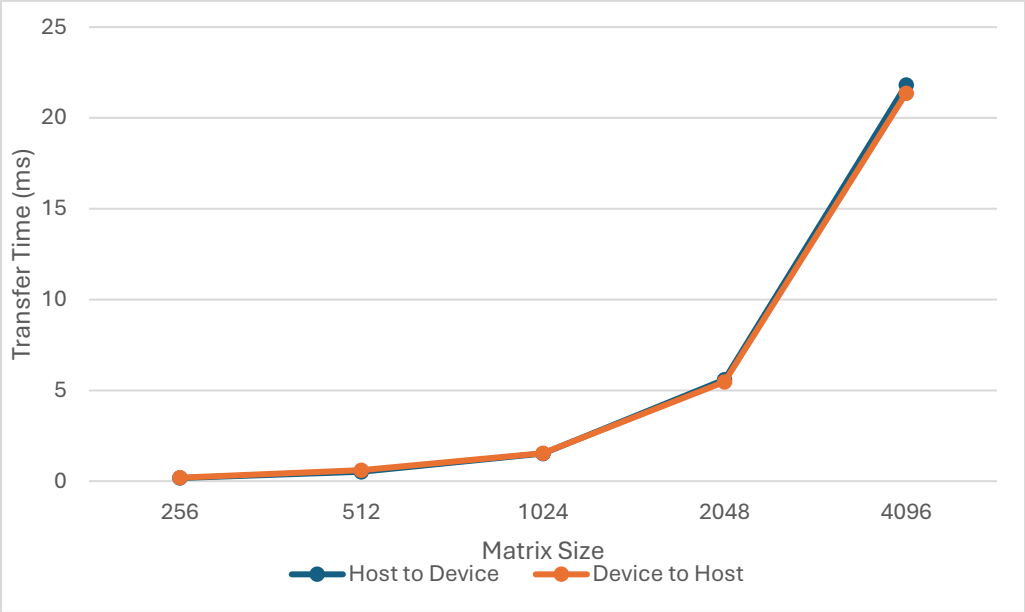

Figure 2: The Effects of Matrix Size on Transfer time (ms) from Host to Device and back

As the graph above illustrates, the Host to Device (H2D) transfer time is relatively the same as Device to Host (D2H). However, upon close inspection H2D seems to be faster for the smaller matrix sizes, and the D2H seems to be faster for larger matrix sizes, though this difference may be insignificant. Seeing no significant difference between H2D and D2H makes sense since the PCIe has equal speeds in either direction.

2.2

Is it always beneficial to offload your matrix multiplication to the device?

> No, it is not always better to offload matrix multiplication to the GPU. If the transfer times from H2D and D2H plus the execution time on the GPU is longer than the CPU execution time, then its best to simply run the calculation on the CPU. However, this is only true for smaller matrices, for even medium sized matrices it is better to run the calculation on the GPU which can be seen from the output. The 256x256 matrix for the output of Part 2.3 takes only 1.75 ms to complete and the transfer time is only 0.17 ms, whereas it takes the CPU 35 ms, as can be seen in the output of Part 2.2. And this holds true as matrix sizes increase, since the GPU is made for massively parallel tasks

2.3

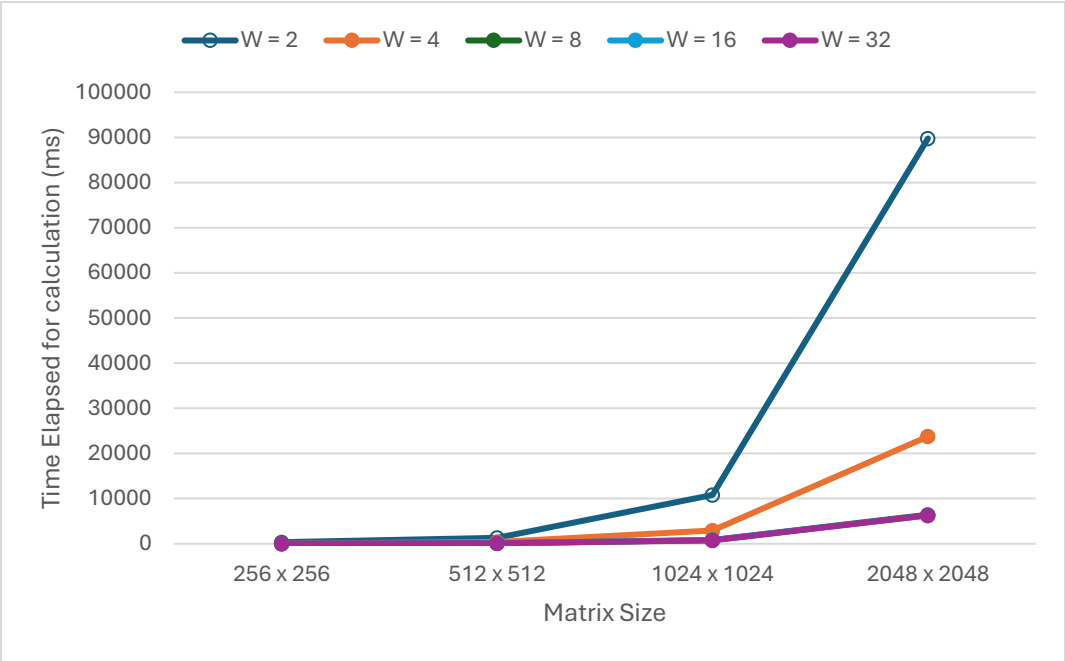

Figure 3: The effect of varying number of blocks/block-width on execution time of Matrices of varying sizes

a) How many times is each element of each input matrix loaded during the execution of the kernel?

In normal matrix multiplication each element of each input matrix should only be loaded once during the execution of the kernel. Since each element of matrix N is loaded once per column in M and each element of matrix M is loaded once per row in N, to complete the matrix multiplication using the naïve approach.

b) What is the floating-point computation to global memory access (CGMA) ratio in each thread? Consider multiply and addition as separate operations and ignore the global memory store at the end. Only count global memory loads towards your off-chip bandwidth.

The equation below describes the formula for the CGMA ratio:

$$CGMA\ ratio = \frac{FLOP}{Total\ \#\ of\ Memory\ Accesses}$$

For our purposes, each element is only read once, thus there are 2 memory accesses, one for each element in any given atomic calculation. In other words, the element from N is selected and an element from M is selected. $Total\ \#\ of\ Memory\ Accesses = 2S$, where S is the size of the matrix. Similarly, there are two floating point operations done, a multiplication and an addition, and thus this leads to $FLOP = 2S$. Plugging in these values leads to a CGMA ratio of 1.

$$CGMA\ ratio = \frac{2S}{2S} = 1$$

Appendix

Images below show Part 1 code, Part 2 code, and full output, in that order (Hint: Check label at top)

MP1_Part1.cu > ...

```cpp
#include <cuda_runtime.h>
#include <stdio.h>
// https://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html

int getCUDACores(cudaDeviceProp p) {
    int coresPerSM = 0;

    // Determine the number of cores per SM based on compute capability information was found at the link below
    // https://developer.nvidia.com/cuda-gpus
    if (p.major == 2) {
        if (p.minor == 1) coresPerSM = 48;
        else coresPerSM = 32;
    }
    else if (p.major == 3) {
        coresPerSM = 192;
    }
    else if (p.major == 5) {
        coresPerSM = 128;
    }
    else if (p.major == 6) {
        if (p.minor == 1 || p.minor == 2) coresPerSM = 128;
        else coresPerSM = 64;
    }
    else if (p.major == 7) {
        coresPerSM = 64;
    }
    else if (p.major == 8) {
        if (p.minor == 0) coresPerSM = 64;
        else coresPerSM = 128;
    }
    else if (p.major == 9) {
        coresPerSM = 128;
    }
    // not including blackwell GPU cause they are too
    else {
        printf("Unknown CUDA architecture: Compute Capability %d.%d\n", p.major, p.minor);
        return 0;
    }

    return p.multiProcessorCount * coresPerSM;
}

int main() {
    int nd;
    cudaGetDeviceCount(&nd);
    printf("Number of CUDA Devices: %d\n", nd);

    if (nd == 0) {
        printf("No CUDA devices found. Exiting...\n");
        return 1;
    }

    for (int i = 0; i < nd; i++) {
        cudaDeviceProp p;
        cudaGetDeviceProperties(&p, i);

        printf("\nDevice %d: %s\n", i, p.name);
        printf("Compute Capability: %d.%d\n", p.major, p.minor);
        printf("Clock Rate: %.2f MHz\n", p.clockRate / 1000.0f);
        printf("Number of Streaming Multiprocessors (SMs): %d\n", p.multiProcessorCount);

        int cudaCores = getCUDACores(p);
        if (cudaCores > 0)
            printf("Total CUDA Cores (approx.): %d\n", cudaCores);
        else
            printf("Could not determine CUDA cores for this device.\n");

        printf("Warp Size: %d\n", p.warpSize);
        printf("Global Memory: %.2f GB\n", p.totalGlobalMem / (1024.0 * 1024.0 * 1024.0));
        printf("Constant Memory: %.2f KB\n", p.totalConstMem / 1024.0);
        printf("Shared Memory per Block: %.2f KB\n", p.sharedMemPerBlock / 1024.0);
        printf("Registers per Block: %d\n", p.regsPerBlock);
        printf("Max Threads per Block: %d\n", p.maxThreadsPerBlock);
        printf("Max Block Dimensions: (%d, %d, %d)\n",
               p.maxThreadsDim[0], p.maxThreadsDim[1], p.maxThreadsDim[2]);
        printf("Max Grid Dimensions: (%d, %d, %d)\n",
               p.maxGridSize[0], p.maxGridSize[1], p.maxGridSize[2]);
    }

    return 0;
}
```

MP1_Part2.cu > ...

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h> // forgot to add earlier
#include <math.h>
#include <time.h>

// Kernel for matrix multiplication on the GPU
__global__ void matMulKernel(const float* M, const float* N, float* P, int width) {
    // Compute row and column indices for this thread
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Out of Bounds check
    if (row < width && col < width) {
        float sum = 0.0f;
        // Perform multiplication
        for (int k = 0; k < width; k++) {
            sum += M[row * width + k] * N[k * width + col];
        }
        // Store the result
        P[row * width + col] = sum;
    }
}

// Single-thread kernel
__global__ void matMulKernelSingleThread(const float* M, const float* N, float* P, int width) {
    if (threadIdx.x == 0 && blockIdx.x == 0 && threadIdx.y == 0 && blockIdx.y == 0) {
        for (int row = 0; row < width; row++) {
            for (int col = 0; col < width; col++) {
                float sum = 0.0f;
                for (int k = 0; k < width; k++) {
                    sum += M[row * width + k] * N[k * width + col];
                }
                P[row * width + col] = sum;
            }
        }
    }
}

// Function for matrix multiplication on the CPU
void matMulCPU(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;
            // Multiply row i of M by column j of N
            for (int k = 0; k < width; k++) {
                sum += M[i * width + k] * N[k * width + j];
            }
            // Store the result in P
            P[i * width + j] = sum;
        }
    }
}

// Comparison function to check correct answer
bool compareArrays(const float* A, const float* B, int size, float tolerance) {
    for (int i = 0; i < size; i++) {
        // Reutrn Falseif difference > tolerance
        if (fabs(A[i] - B[i]) > tolerance) {
            return false;
        }
    }
    return true;
}
// Kernel execution time (data transfer not included)
void measureKernelTime(const float* dM, const float* dN, float* dP,
    int width, int blockWidth, float &kernelTimeMs) {
    // Create the dim3 block and grid
    dim3 block(blockWidth, blockWidth);
    dim3 grid((width + blockWidth - 1) / blockWidth, (width + blockWidth - 1) / blockWidth);

    // Create CUDA events to record time taken
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Record the start event, launch kernel
    cudaEventRecord(start);
    matMulKernel<<<grid, block>>>(dM, dN, dP, width);
    cudaEventRecord(stop);

    // Synchronize to make sure kernel finished, measure time
    // Was causing error (may not be needed)
    cudaEventSynchronize(stop);

    // Calculate final value
    cudaEventElapsedTime(&kernelTimeMs, start, stop);

    // Free memory
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
}
```

MP1_Part2.cu > ...

```
int main()
    // Part 2.1: Transfer Times
    const int sizes1[] = {256, 512, 1024, 2048, 4096};
    const int numSizes1 = sizeof(sizes1) / sizeof(int);

    printf("--- Part 2.1: H to D and D to H Transfer Times ---\n");
    printf("Matrix Sizes: 256, 512, 1024, 2048, 4096\n\n");

    // Arrays to store transfer times
    float hToDTimes[numSizes1];
    float dToHTimes[numSizes1];

    // Iterate through varying sizes
    for (int idx = 0; idx < numSizes1; idx++) {
        int width = sizes1[idx];
        size_t bytes = width * (size_t)width * sizeof(float);

        // Allocate host memory
        float* hM = (float*)malloc(bytes);
        float* hN = (float*)malloc(bytes);
        float* hP = (float*)malloc(bytes);

        // Initialize data
        srand(0);
        for (int i = 0; i < width * width; i++) {
            hM[i] = (float)(rand() % 10);
            hN[i] = (float)(rand() % 10);
        }

        // Allocate device memory
        float* dM; float* dN; float* dP;
        cudaMalloc((void**)&dM, bytes);
        cudaMalloc((void**)&dN, bytes);
        cudaMalloc((void**)&dP, bytes);

        // Measure Host to Device transfer time
        cudaEvent_t startHtoD, stopHtoD;
        cudaEventCreate(&startHtoD);
        cudaEventCreate(&stopHtoD);
        cudaEventRecord(startHtoD);
        cudaMemcpy(dM, hM, bytes, cudaMemcpyHostToDevice);
        cudaMemcpy(dN, hN, bytes, cudaMemcpyHostToDevice);
        cudaEventRecord(stopHtoD);
        cudaEventSynchronize(stopHtoD); // for error

        float timeHtoD = 0.0f;
        cudaEventElapsedTime(&timeHtoD, startHtoD, stopHtoD);
        hToDTimes[idx] = timeHtoD;

        // Measure Device to Host transfer time
        // Typically you'd measure for P but using N and M for simplicity
        cudaEvent_t startDtoH, stopDtoH;
        cudaEventCreate(&startDtoH);
        cudaEventCreate(&stopDtoH);
        cudaEventRecord(startDtoH);
        cudaMemcpy(hP, dM, bytes, cudaMemcpyDeviceToHost);
        cudaMemcpy(hP, dN, bytes, cudaMemcpyDeviceToHost);
        cudaEventRecord(stopDtoH);
        cudaEventSynchronize(stopDtoH);

        float timeDtoH = 0.0f;
        cudaEventElapsedTime(&timeDtoH, startDtoH, stopDtoH);
        dToHTimes[idx] = timeDtoH;

        // Free memory
        cudaFree(dM);
        cudaFree(dN);
        cudaFree(dP);
        free(hM);
        free(hN);
        free(hP);
        cudaEventDestroy(startHtoD);
        cudaEventDestroy(stopHtoD);
        cudaEventDestroy(startDtoH);
        cudaEventDestroy(stopDtoH);
    }

    // Print results for Part 2.1
    printf("Host to Device Transfer Times (ms) by Matrix Size:\n");
    for (int i = 0; i < numSizes1; i++) {
        printf("  Size %d x %d : %f ms\n", sizes1[i], sizes1[i], hToDTimes[i]);
    }
    printf("\nDevice to Host Transfer Times (ms) by Matrix Size:\n");
    for (int i = 0; i < numSizes1; i++) {
        printf("  Size %d x %d : %f ms\n", sizes1[i], sizes1[i], dToHTimes[i]);
    }
    printf("\n");

    // Part 2.2
    const int sizes2[] = {256, 512, 1024};
    const int numSizes2 = sizeof(sizes2) / sizeof(int);

    printf("--- Part 2.2 : CPU vs. GPU (Single Thread) ---\n");
    printf("MatrixSizes: 256, 512, 1024\n\n");

    for (int idx = 0; idx < numSizes2; idx++) {
        int width = sizes2[idx];
        size_t bytes = width * (size_t)width * sizeof(float);

        float* hM = (float*)malloc(bytes);
        float* hN = (float*)malloc(bytes);
        float* hP = (float*)malloc(bytes);
        float* hRef = (float*)malloc(bytes);

```

MP1_Part2.cu > ...

```
96  int main()
192     for (int idx = 0; idx < numSizes2; idx++) {
200
201         srand(0);
202         for (int i = 0; i < width * width; i++) {
203             hM[i] = (float)(rand() % 10);
204             hN[i] = (float)(rand() % 10);
205         }
206
207         float *dM, *dN, *dP;
208         cudaMalloc((void**)&dM, bytes);
209         cudaMalloc((void**)&dN, bytes);
210         cudaMalloc((void**)&dP, bytes);
211
212         cudaEvent_t startHtoD, stopHtoD;
213         cudaEventCreate(&startHtoD);
214         cudaEventCreate(&stopHtoD);
215         cudaEventRecord(startHtoD);
216         cudaMemcpy(dM, hM, bytes, cudaMemcpyHostToDevice);
217         cudaMemcpy(dN, hN, bytes, cudaMemcpyHostToDevice);
218         cudaEventRecord(stopHtoD);
219         cudaEventSynchronize(stopHtoD);
220
221         float hToD_ms = 0.0f;
222         cudaEventElapsedTime(&hToD_ms, startHtoD, stopHtoD);
223
224         // Assign only 1 block with 1 thread as instructed
225         dim3 block(1, 1);
226         dim3 grid(1, 1);
227
228         cudaEvent_t startKernel, stopKernel;
229         cudaEventCreate(&startKernel);
230         cudaEventCreate(&stopKernel);
231
232         // GPU
233         cudaEventRecord(startKernel);
234         matMulKernelSingleThread<<<grid, block>>>(dM, dN, dP, width);
235         cudaEventRecord(stopKernel);
236         cudaEventSynchronize(stopKernel);
237
238         float gpuKernel_ms = 0.0f;
239         cudaEventElapsedTime(&gpuKernel_ms, startKernel, stopKernel);
240
241         cudaEvent_t startDtoH, stopDtoH;
242         cudaEventCreate(&startDtoH);
243         cudaEventCreate(&stopDtoH);
244
245         cudaEventRecord(startDtoH);
246         cudaMemcpy(hP, dP, bytes, cudaMemcpyDeviceToHost);
247         cudaEventRecord(stopDtoH);
248         cudaEventSynchronize(stopDtoH);
249
250         float dToH_ms = 0.0f;
251         cudaEventElapsedTime(&dToH_ms, startDtoH, stopDtoH);
252
253         // CPU time
254         clock_t cpuStart = clock();
255         matMulCPU(hM, hN, hRef, width);
256         clock_t cpuEnd = clock();
257         float cpu_ms = 1000.0f * (float)(cpuEnd - cpuStart) / CLOCKS_PER_SEC;
258
259         // Getting values for Ignoring transfer vs. including transfer
260         float gpuTotalNoTransfer = gpuKernel_ms;               // GPU only kernel
261         float gpuTotalWithTransfer = gpuKernel_ms + hToD_ms + dToH_ms;
262
263         // Compare correctness tolerance chosen manually
264         bool pass = compareArrays(hRef, hP, width * width, 1e-3f);
265
266         printf("Matrix Size %d x %d\n", width, width);
267         printf("  CPU Time (ms)                          : %f\n", cpu_ms);
268         printf("  GPU Time (1 block,1 thread) (ms) : %f (NO Transfer), %f (WITH Transfer)\n",
269                gpuTotalNoTransfer, gpuTotalWithTransfer);
270         printf("    Transfer Times: H to D = %f ms, D to H = %f ms\n", hToD_ms, dToH_ms);
271         printf("    %s\n\n", pass ? "Test PASSED" : "Test FAILED");
272
273         // Clean up
274         cudaFree(dM);
275         cudaFree(dN);
276         cudaFree(dP);
277         free(hM);
278         free(hN);
279         free(hP);
280         free(hRef);
281         cudaEventDestroy(startHtoD);
282         cudaEventDestroy(stopHtoD);
283         cudaEventDestroy(startKernel);
284         cudaEventDestroy(stopKernel);
285         cudaEventDestroy(startDtoH);
286         cudaEventDestroy(stopDtoH);
287     }
288
289     // Part 3.3: Vary block width and only measure kernel times
290     printf("--- Part 3.3: Kernel Times with varying Matrix Size & Block Width---\n");
291     printf("MatrixSizes: 256, 512, 1024, 2048, 4096  |  BlockWidth: 2,4,8,16,32\n\n");
292
293     int blockWidths[5] = {2, 4, 8, 16, 32};
294     int sizes3[] = {256, 512, 1024, 2048, 4096};
295     int numSizes3 = 5;
296
```

MP1_Part2.cu > main()

```
96  int main()
295     int numSizes3 = 5;
296
297     // Print a table of times for each (matrixSize, blockWidth) using for loop for ease
298     for (int bwIdx = 0; bwIdx < 5; bwIdx++) {
299         int bWidth = blockWidths[bwIdx];
300         printf("BlockWidth = %d\n", bWidth);
301         for (int sIdx = 0; sIdx < numSizes3; sIdx++) {
302             int width = sizes3[sIdx];
303             size_t bytes = width * (size_t)width * sizeof(float);
304
305             float* hM = (float*)malloc(bytes);
306             float* hN = (float*)malloc(bytes);
307             float* hP = (float*)malloc(bytes);
308
309             srand(0);
310             for (int i = 0; i < width * width; i++) {
311                 hM[i] = (float)(rand() % 10);
312                 hN[i] = (float)(rand() % 10);
313             }
314
315             float *dM, *dN, *dP;
316             cudaMalloc((void**)&dM, bytes);
317             cudaMalloc((void**)&dN, bytes);
318             cudaMalloc((void**)&dP, bytes);
319
320             // Transfer time is ignored
321             cudaMemcpy(dM, hM, bytes, cudaMemcpyHostToDevice);
322             cudaMemcpy(dN, hN, bytes, cudaMemcpyHostToDevice);
323
324             // Measure kernel time
325             float kernelMs = 0.0f;
326             measureKernelTime(dM, dN, dP, width, bWidth, kernelMs);
327
328             printf("  Size %d x %d -> Kernel Time = %f ms\n", width, width, kernelMs);
329
330             // Free memory
331             cudaFree(dM);
332             cudaFree(dN);
333             cudaFree(dP);
334             free(hM);
335             free(hN);
336             free(hP);
337         }
338         printf("\n");
339     }
340
341     return 0;
342 }
```

Outputs.txt

```
Part 1:
Device 0: NVIDIA T600
Compute Capability: 7.5
Clock Rate: 1335.00 MHz
Number of Streaming Multiprocessors (SMs): 10
Total CUDA Cores (approx.): 640
Warp Size: 32
Global Memory: 4.00 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)

Part 2:
--- Part 2.1: H to D and D to H Transfer Times ---
MatrixSizes: 256, 512, 1024, 2048, 4096

Host to Device Transfer Times (ms) by Matrix Size:
  Size 256 x 256 : 0.173728 ms
  Size 512 x 512 : 0.514912 ms
  Size 1024 x 1024 : 1.522336 ms
  Size 2048 x 2048 : 5.598752 ms
  Size 4096 x 4096 : 21.819136 ms

Device to Host Transfer Times (ms) by Matrix Size:
  Size 256 x 256 : 0.191232 ms
  Size 512 x 512 : 0.601984 ms
  Size 1024 x 1024 : 1.534912 ms
  Size 2048 x 2048 : 5.465600 ms
  Size 4096 x 4096 : 21.367456 ms

--- Part 2.2 : CPU vs. GPU (Single Thread) ---
MatrixSizes: 256, 512, 1024

Matrix Size 256 x 256
  CPU Time (ms)                  : 35.000000
  GPU Time (1 block,1 thread) (ms) : 8532.343750 (NO Transfer), 8532.598633 (WITH Transfer)
    Transfer Times: H to D = 0.156704 ms, D to H = 0.098944 ms
    Test PASSED

Matrix Size 512 x 512
  CPU Time (ms)                  : 475.000000
  GPU Time (1 block,1 thread) (ms) : 78453.343750 (NO Transfer), 78454.273438 (WITH Transfer)
    Transfer Times: H to D = 0.550944 ms, D to H = 0.372160 ms
    Test PASSED

Matrix Size 1024 x 1024
  CPU Time (ms)                  : 3718.000000
  GPU Time (1 block,1 thread) (ms) : 656019.062500 (NO Transfer), 656022.812500 (WITH Transfer)
    Transfer Times: H to D = 1.570624 ms, D to H = 2.213408 ms
    Test PASSED

--- Part 3.3: Kernel Times with varying Matrix Size & Block Width---
MatrixSizes: 256, 512, 1024, 2048, 4096  |  BlockWidth: 2,4,8,16,32

BlockWidth = 2
  Size 256 x 256 -> Kernel Time = 164.616379 ms
  Size 512 x 512 -> Kernel Time = 353.023529 ms
  Size 1024 x 1024 -> Kernel Time = 1286.574219 ms
  Size 2048 x 2048 -> Kernel Time = 10793.482422 ms
  Size 4096 x 4096 -> Kernel Time = 89778.617188 ms

BlockWidth = 4
  Size 256 x 256 -> Kernel Time = 4.584320 ms
  Size 512 x 512 -> Kernel Time = 38.927265 ms
  Size 1024 x 1024 -> Kernel Time = 366.748840 ms
  Size 2048 x 2048 -> Kernel Time = 2919.217529 ms
  Size 4096 x 4096 -> Kernel Time = 23755.808594 ms

BlockWidth = 8
  Size 256 x 256 -> Kernel Time = 1.670912 ms
  Size 512 x 512 -> Kernel Time = 12.938912 ms
  Size 1024 x 1024 -> Kernel Time = 87.001022 ms
  Size 2048 x 2048 -> Kernel Time = 791.867615 ms
  Size 4096 x 4096 -> Kernel Time = 6442.567871 ms

BlockWidth = 16
  Size 256 x 256 -> Kernel Time = 1.344320 ms
  Size 512 x 512 -> Kernel Time = 11.579136 ms
  Size 1024 x 1024 -> Kernel Time = 104.591614 ms
  Size 2048 x 2048 -> Kernel Time = 788.934998 ms
  Size 4096 x 4096 -> Kernel Time = 6290.804688 ms

BlockWidth = 32
  Size 256 x 256 -> Kernel Time = 1.751808 ms
  Size 512 x 512 -> Kernel Time = 13.951296 ms
  Size 1024 x 1024 -> Kernel Time = 89.537376 ms
  Size 2048 x 2048 -> Kernel Time = 760.267029 ms
  Size 4096 x 4096 -> Kernel Time = 6298.536133 ms
```