# Part 1

Code was written to print out the proper parameters, the output is in Figure 1. The full code can be found in the Appendix.

A computer screen shot of a black screen

Description automatically generated

Figure 1 Output Part 1

# Part 2

## Section 1

Code was written to transfer the matrices from the host to device. The results for different matrix sizes can be found below in \_\_\_\_. Plotting these results can be found in \_\_\_\_, below the table of results.

Example of output is seen in \_\_\_ below:

A black screen with white text

Description automatically generated

|  |  |
| --- | --- |
| **Matrix Size** | **Time (ms)** |
| 100 | 0.075392 |
| 250 | 0.101120 |
| 500 | 0.217952 |
| 1000 | 0.545792 |
| 1500 | 1.036960 |

PLOT DATA!!!

The code was modified to also transfer data back from device to host, results seen in \_\_\_\_ below.

|  |  |  |
| --- | --- | --- |
| **Matrix Size** | **Time (ms) Host to Device** | **Time (ms) Device to Host** |
| 100 | 0.082720 | 0.031360 |
| 250 | 0.068384 | 0.049920 |
| 500 | 0.202464 | 0.178208 |
| 1000 | 0.496544 | 0.507104 |
| 1500 | 0.971104 | 1.040672 |

PLOT DATA!!!

You can see that there is a distinct consistent difference between transfer times comparing host to device and device to host. Device to host is consistently slower. This makes sense, as the PCIe transfer bus do not operate in a traditionally parallel mode. For this reason, since the CPU is better at handling series operations, it would be faster.

Code for Part 2 Section 1 can be found in the Appendix.

## Section 2

In section 2, code needed to be modified to perform matrix multiplication on the CPU and GPU. At first, a BLOCK\_WIDTH of 1 (1 thread) was used, along with the number of blocks being 1. Sample output seen in \_\_\_\_. Table containing results is below in \_\_\_\_\_.

A screenshot of a computer

Description automatically generated

Note: Time in ms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Matrix Size** | **GPU Time (mul)** | **GPU Time (all)** | **CPU Time (mul)** | **GPU Time (all)** |
| 100 | 0.282720 |  | 1.512992 |  |
| 250 | 0.457952 |  | 22.819136 |  |
| 500 | 5.199040 |  | 191.184006 |  |
| 1000 | 1.444320 |  | 1577.490234 |  |
| 1500 | 14.211872 |  | 5558.251953 |  |

# Appendix

## Part 1 Code

#include <cuda\_runtime.h>

#include <stdio.h>

// Helper function to convert compute capability to the number of cores

int ConvertSMVer2Cores(int major, int minor) {

int cores;

switch ((major << 4) + minor) {

case 0x10:

cores = 8;

break;

case 0x11:

case 0x12:

cores = 8;

break;

case 0x13:

cores = 32;

break;

case 0x20:

cores = 32;

break;

default:

cores = 0;

break;

}

return cores;

}

int main() {

cudaSetDevice(0); // Set the device to GPU 0 (or the appropriate GPU index)

int num\_devices;

cudaGetDeviceCount(&num\_devices);

if (num\_devices == 0) {

fprintf(stderr, "No CUDA devices found.\n");

return 1;

}

for (int device\_id = 0; device\_id < num\_devices; ++device\_id) {

cudaDeviceProp device\_prop;

cudaError\_t cuda\_status = cudaGetDeviceProperties(&device\_prop, device\_id);

if (cuda\_status != cudaSuccess) {

fprintf(stderr, "Error: cudaGetDeviceProperties failed with error code %d\n", cuda\_status);

return 1;

}

printf("Device %d Information:\n", device\_id);

printf("Name: %s\n", device\_prop.name);

printf("Compute Capability: %d.%d\n", device\_prop.major, device\_prop.minor);

printf("Clock Rate: %d kHz\n", device\_prop.clockRate);

printf("Number of SM (Streaming Multiprocessors): %d\n", device\_prop.multiProcessorCount);

printf("Number of Cores per SM: %d\n", ConvertSMVer2Cores(device\_prop.major, device\_prop.minor) \* device\_prop.multiProcessorCount);

printf("Warp Size: %d\n", device\_prop.warpSize);

printf("Global Memory Size: %zu bytes\n", device\_prop.totalGlobalMem);

printf("Constant Memory Size: %zu bytes\n", device\_prop.totalConstMem);

printf("Shared Memory Size per Block: %zu bytes\n", device\_prop.sharedMemPerBlock);

printf("Registers per Block: %d\n", device\_prop.regsPerBlock);

printf("Max Threads per Block: %d\n", device\_prop.maxThreadsPerBlock);

printf("Max Size of Each Dimension of a Block: (%d, %d, %d)\n",

device\_prop.maxThreadsDim[0], device\_prop.maxThreadsDim[1], device\_prop.maxThreadsDim[2]);

printf("Max Size of Each Dimension of a Grid: (%d, %d, %d)\n",

device\_prop.maxGridSize[0], device\_prop.maxGridSize[1], device\_prop.maxGridSize[2]);

printf("\n");

}

return 0;

}

## Part 2 Section 1 Code

// Jacob Badali 20290739

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <math.h>

#include <stdlib.h>

#define Threads\_per\_block 128

#define WIDTH (1500) //Not sure what this size means, will find out later. wondering why it isn't like [][]...

#define BLOCK\_WIDTH 32

//Multiplication kernel function

\_\_global\_\_ void mulKernel(int\* M, int\* N, int\* P, int size) {

int rows = blockIdx.y \* blockDim.y + threadIdx.y;

int cols = blockIdx.x \* blockDim.x + threadIdx.x;

if (rows < size && cols < size) {

float temp\_sum = 0.0;

for (int i = 0; i < size; i++) {

temp\_sum += M[rows \* size + i] \* N[i \* size + cols];

}

P[rows \* size + cols] = temp\_sum;

}

}

void mulDevice(float\* A, float\* B, float\* C, int size) {

for (int i = 0; i < size; ++i) {

for (int j = 0; j < size; ++j) {

float temp = 0;

for (int k = 0; k < size; ++k) {

temp += A[i \* size + k] \* B[k \* size + j];

}

C[i \* size + j] = temp;

}

}

}

int main()

{

float\* d\_M = 0;

float\* d\_N = 0;

float\* d\_P = 0;

float\* h\_M;

float\* h\_N;

float\* h\_P;

//int WIDTH[5] = { 100, 250, 500, 1000, 2500 }; // Initialize WIDTH array here

//for (int i = 0; i < 5; i++) {

int size = WIDTH \* WIDTH \* sizeof(float);

cudaMallocHost((void\*\*)&h\_M, size);

cudaMallocHost((void\*\*)&h\_N, size);

cudaMallocHost((void\*\*)&h\_P, size);

int NumBlocks = WIDTH / BLOCK\_WIDTH;

if (WIDTH % BLOCK\_WIDTH) NumBlocks++;

dim3 dimGrid(NumBlocks, NumBlocks);

dim3 dimBlock(BLOCK\_WIDTH, BLOCK\_WIDTH);

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

float elapsedTime\_DevToHost;

float elapsedTime\_HostToDev;

//Allocate appropriate memory size for each array

cudaMalloc((void\*\*)&d\_M, size);

cudaMalloc((void\*\*)&d\_N, size);

cudaMalloc((void\*\*)&d\_P, size);

//fill host matrices

for (int k = 0; k < WIDTH; k++) {

for (int j = 0; j < WIDTH; j++) {

h\_M[k \* WIDTH + j] = ((float)rand() / RAND\_MAX) \* 100.0; // fill with rand values from 0-100

h\_N[k \* WIDTH + j] = ((float)rand() / RAND\_MAX) \* 100.0;

h\_P[k \* WIDTH + j] = 0;

}

}

/\*

//Host matrix multiplication

for (int i = 0; i < num\_row; i++) {

for (int j = 0; j < num\_col; j++) {

h\_P[i][j] = 0;

for (int k = 0; k < num\_row; k++) {

h\_P[i][j] += h\_M[i][k] \* h\_N[k][j];

}

}

}

\*/

//Cpy to dev, timer

cudaEventRecord(start, 0);

cudaMemcpy(d\_M, h\_M, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_N, h\_N, size, cudaMemcpyHostToDevice);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime\_HostToDev, start, stop);

printf("Transfer Host to Device, size[%d]: %f ms |", WIDTH, elapsedTime\_HostToDev);

cudaEventRecord(start, 0);

cudaMemcpy(h\_M, d\_M, size, cudaMemcpyDeviceToHost);

cudaMemcpy(h\_N, d\_N, size, cudaMemcpyDeviceToHost);

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&elapsedTime\_DevToHost, start, stop);

printf("| Transfer Device to Host, size[%d]: %f ms\n", WIDTH, elapsedTime\_DevToHost);

//Device Matrix multiplication

//mulKernel << <dimBlock, dimGrid, 0, 0 >> > (d\_M, d\_N, d\_P, size);

//Cpy to dev, timer

cudaFree(d\_M);

cudaFree(d\_N);

cudaFree(d\_P);

}

//}

## Part 2 Section 2 Code