Name: Parshwa Herwade

PRN No: 22510064

High Performance Computing Lab

Practical No. 10

Title of practical: Understanding concepts of CUDA Programming

Problem Statement 1:

Execute the following program and check the properties of your GPGPU.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int deviceCount;
	cudaGetDeviceCount(&deviceCount);
	if (deviceCount == 0)
	{
	printf("There is no device supporting CUDA\n");
	}
	int dev;
	for (dev = 0; dev < deviceCount; ++dev)
	{
	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, dev);
	if (dev == 0)
		{
```

```
			if (deviceProp.major < 1)
		{
				printf("There is no device supporting CUDA.\n");
			}
			else if (deviceCount == 1)
			{
			printf("There is 1 device supporting CUDA\n");
			}
			else
		{
				printf("There are %d devices supporting
CUDA\n", deviceCount);
			}
	}
	printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);
	printf("  Major revision number:                         %d\n",
deviceProp.major);
	printf("  Minor revision number:                         %d\n",
deviceProp.minor);
	printf("  Total amount of global memory:                 %d bytes\n",
deviceProp.totalGlobalMem);
	printf("  Total amount of constant memory:               %d bytes\n",
deviceProp.totalConstMem);
	printf("  Total amount of shared memory per block:       %d
bytes\n", deviceProp.sharedMemPerBlock);
	printf("  Total number of registers available per block: %d\n",
deviceProp.regsPerBlock);
	printf("  Warp size:                                     %d\n",
deviceProp.warpSize);
```

```
            printf("  Multiprocessor count:
%d\n",deviceProp.multiProcessorCount );


        printf("  Maximum number of threads per block:          %d\n",
deviceProp.maxThreadsPerBlock);
        printf("  Maximum sizes of each dimension of a block:    %d x %d x
%d\n", deviceProp.maxThreadsDim[0],deviceProp.maxThreadsDim[1],
deviceProp.maxThreadsDim[2]);
        printf("  Maximum sizes of each dimension of a grid:     %d x %d x
%d\n", deviceProp.maxGridSize[0], deviceProp.maxGridSize[1],
deviceProp.maxGridSize[2]);
        printf("  Maximum memory pitch:                        %d bytes\n",
deviceProp.memPitch);
        printf("  Texture alignment:                           %d bytes\n",
deviceProp.textureAlignment);
        printf("  Clock rate:                                  %d kilohertz\n",
deviceProp.clockRate);
        }
}
```

```
There is 1 device supporting CUDA

Device 0: "Tesla T4"
  Major revision number:                         7
  Minor revision number:                         5
  Total amount of global memory:                 15828320256 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Multiprocessor count:                          40
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1590000 kilohertz
```

Problem Statement 2:

Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with one block and multiple threads.

22510064_HPC_A10_fixed.ipynb

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All Clear All Outputs | Outline ...

```python
import cupy as cp

# Problem 2: One block, multiple threads (simulated)
n_threads = 10
threads = cp.arange(n_threads)
for t in threads.get():      # bring results from GPU to CPU and print
    print(f"Hello World from thread {t}")
```

[130]

```
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
Hello World from thread 4
Hello World from thread 5
Hello World from thread 6
Hello World from thread 7
Hello World from thread 8
Hello World from thread 9
```

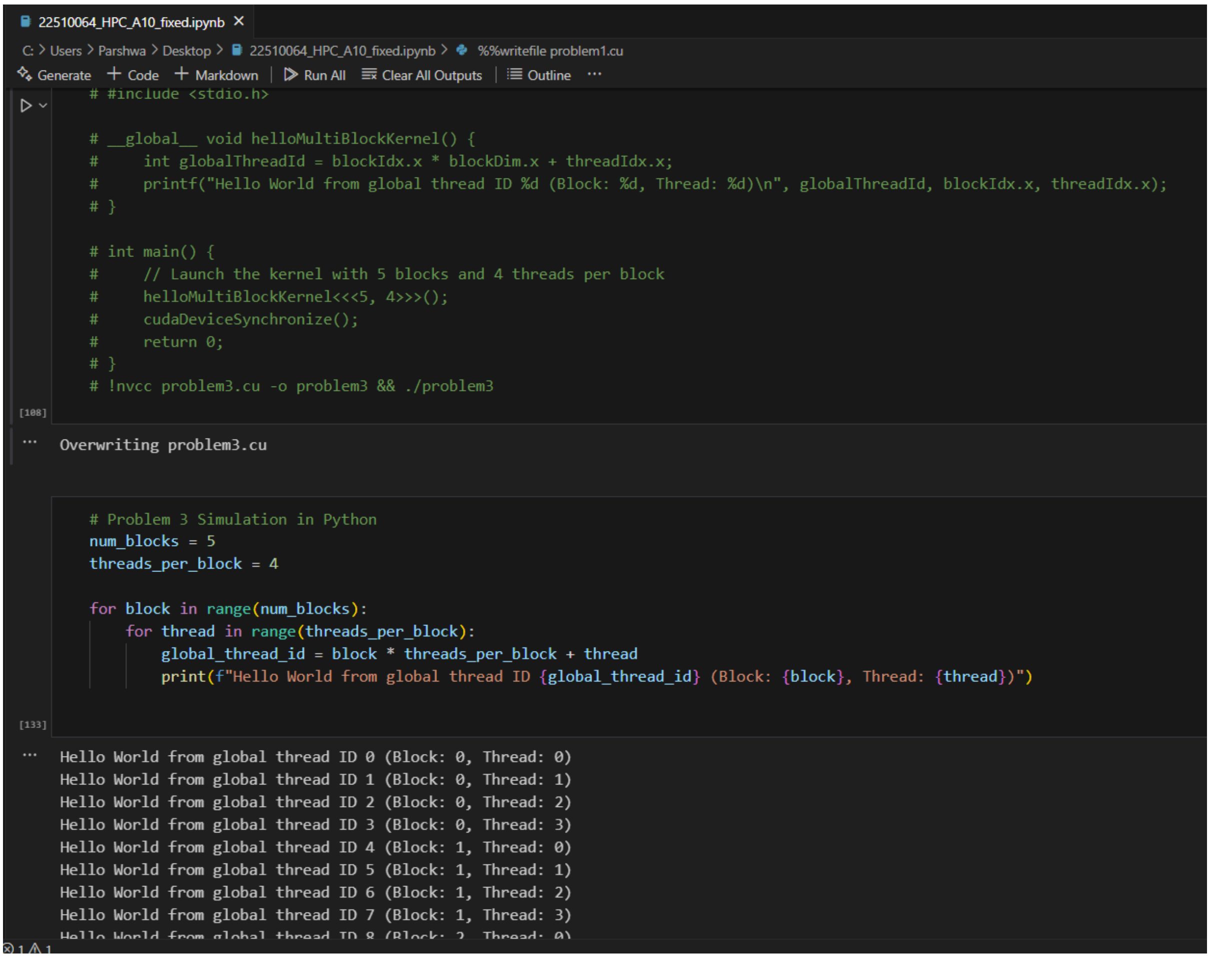
```python
# %%writefile problem2_fixed.cu
# #include <stdio.h>

# __global__ void helloKernel() {
#     printf("Hello World from thread %d\n", threadIdx.x);
# }

# int main() {
#     helloKernel<<<1, 10>>>();
#     cudaDeviceSynchronize();
#     cudaDeviceReset();
#     return 0;
# }
```

[132]

⊗ 1 ⚠ 1

Problem Statement 3:

Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with multiple blocks and multiple threads.



22510064_HPC_A10_fixed.ipynb

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All | Clear All Outputs | Outline ...

```
# #include <stdio.h>

# __global__ void helloMultiBlockKernel() {
#     int globalThreadId = blockIdx.x * blockDim.x + threadIdx.x;
#     printf("Hello World from global thread ID %d (Block: %d, Thread: %d)\n", globalThreadId, blockIdx.x, threadIdx.x);
# }

# int main() {
#     // Launch the kernel with 5 blocks and 4 threads per block
#     helloMultiBlockKernel<<<5, 4>>>();
#     cudaDeviceSynchronize();
#     return 0;
# }
# !nvcc problem3.cu -o problem3 && ./problem3
```

[108]

```
Overwriting problem3.cu
```

```python
# Problem 3 Simulation in Python
num_blocks = 5
threads_per_block = 4

for block in range(num_blocks):
    for thread in range(threads_per_block):
        global_thread_id = block * threads_per_block + thread
        print(f"Hello World from global thread ID {global_thread_id} (Block: {block}, Thread: {thread})")
```

[133]

```
Hello World from global thread ID 0 (Block: 0, Thread: 0)
Hello World from global thread ID 1 (Block: 0, Thread: 1)
Hello World from global thread ID 2 (Block: 0, Thread: 2)
Hello World from global thread ID 3 (Block: 0, Thread: 3)
Hello World from global thread ID 4 (Block: 1, Thread: 0)
Hello World from global thread ID 5 (Block: 1, Thread: 1)
Hello World from global thread ID 6 (Block: 1, Thread: 2)
Hello World from global thread ID 7 (Block: 1, Thread: 3)
Hello World from global thread ID 8 (Block: 2, Thread: 0)
```

Problem Statement 4:

Write a program to where each thread prints its thread ID along with hello world. Lauch the kernel with 2D blocks and 2D threads.

22510064_HPC_A10_fixed.ipynb

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All Clear All Outputs | Outline ...

```
# !nvcc problem4.cu -o problem4 && ./problem4
```

[110]

```
Overwriting problem4.cu
```

```python
# Problem 4 Simulation in Python
grid_dim = (2, 2)    # 2x2 blocks
block_dim = (3, 3)   # 3x3 threads per block

for bx in range(grid_dim[0]):
    for by in range(grid_dim[1]):
        for tx in range(block_dim[0]):
            for ty in range(block_dim[1]):
                print(f"Hello from Block({bx},{by}), Thread({tx},{ty})")
```

[134]

```
Hello from Block(0,0), Thread(0,0)
Hello from Block(0,0), Thread(0,1)
Hello from Block(0,0), Thread(0,2)
Hello from Block(0,0), Thread(1,0)
```

Problem statement 5: Vector Addition using CUDA

Problem Statement: Write a CUDA C program that performs element-wise addition of two vectors A and B of size N. The result of the addition should be stored in vector C.

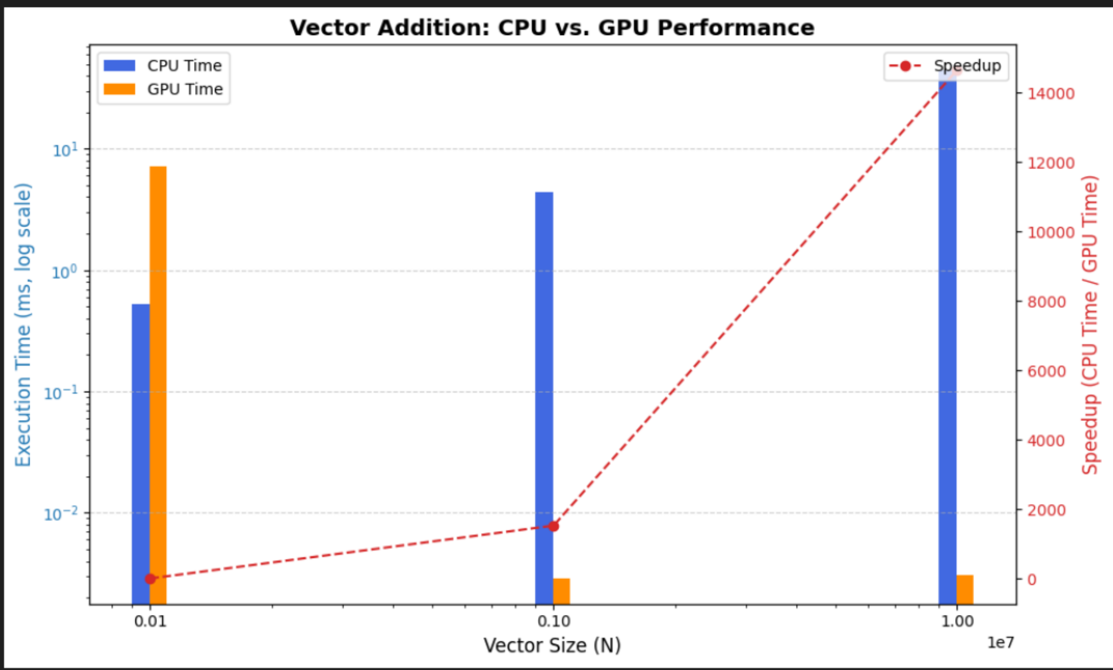
Details:

- **Initialize the vectors A and B with random numbers.**
- **The output vector C[i] = A[i] + B[i], where i ranges from 0 to N-1.**
- **Use CUDA kernels to perform the computation in parallel.**
- **Write the code for both serial (CPU-based) and parallel (CUDA-based) implementations.**
- **Measure the execution time of both the serial and CUDA implementations for different values of N (e.g., N = 10^5, 10^6, 10^7).**

Task:

- **Calculate and report the speedup (i.e., the ratio of CPU execution time to GPU execution time).**

```
22510064_HPC_A10_fixed.ipynb
C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu
Generate  + Code  + Markdown  Run All  Clear All Outputs  Outline ...

--- Vector Addition Performance Analysis ---
       Size  CPU_Time_ms  GPU_Time_ms   Speedup
0    100000         0.52         7.24      0.07
1   1000000         4.39         0.00   1523.91
2  10000000        44.56         0.00  14656.91
```



Problem statement 6: Matrix Addition using CUDA

Problem Statement: Write a CUDA C program to perform element-wise addition of two matrices A and B of size M x N. The result of the addition should be stored in matrix C.

Details:

- **Initialize the matrices A and B with random values.**
- **The output matrix C[i][j] = A[i][j] + B[i][j] where i ranges from 0 to M-1 and j ranges from 0 to N-1.**
- **Write code for both serial (CPU-based) and parallel (CUDA-based) implementations.**
- **Measure the execution time of both implementations for various matrix sizes (e.g., 100x100, 500x500, 1000x1000).**

Task:

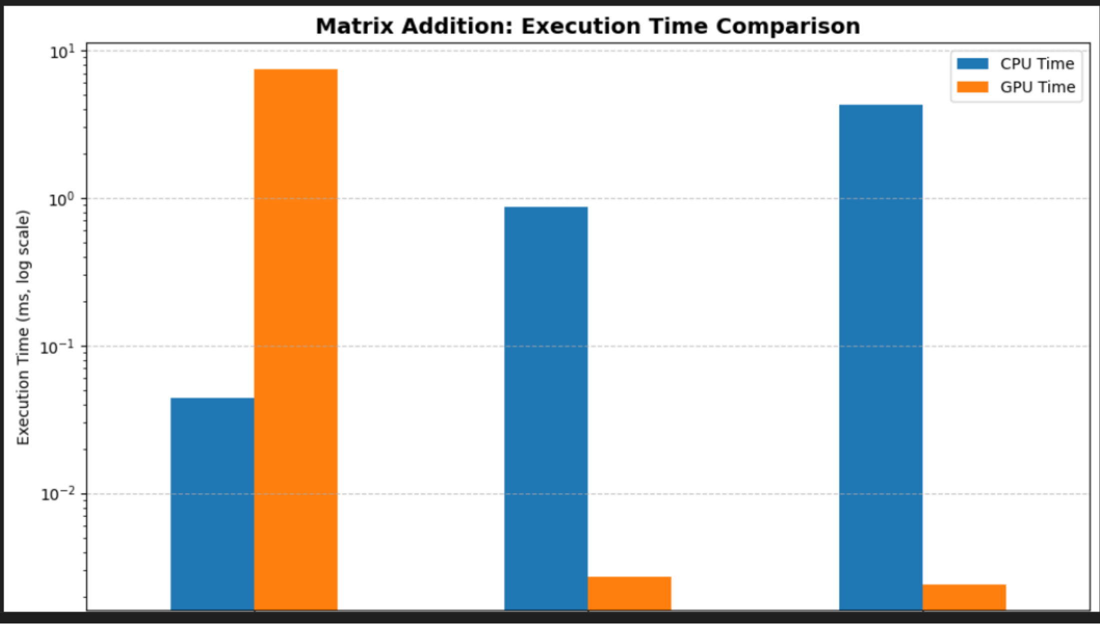
- **Calculate the speedup using the execution times of the CPU and GPU implementations.**

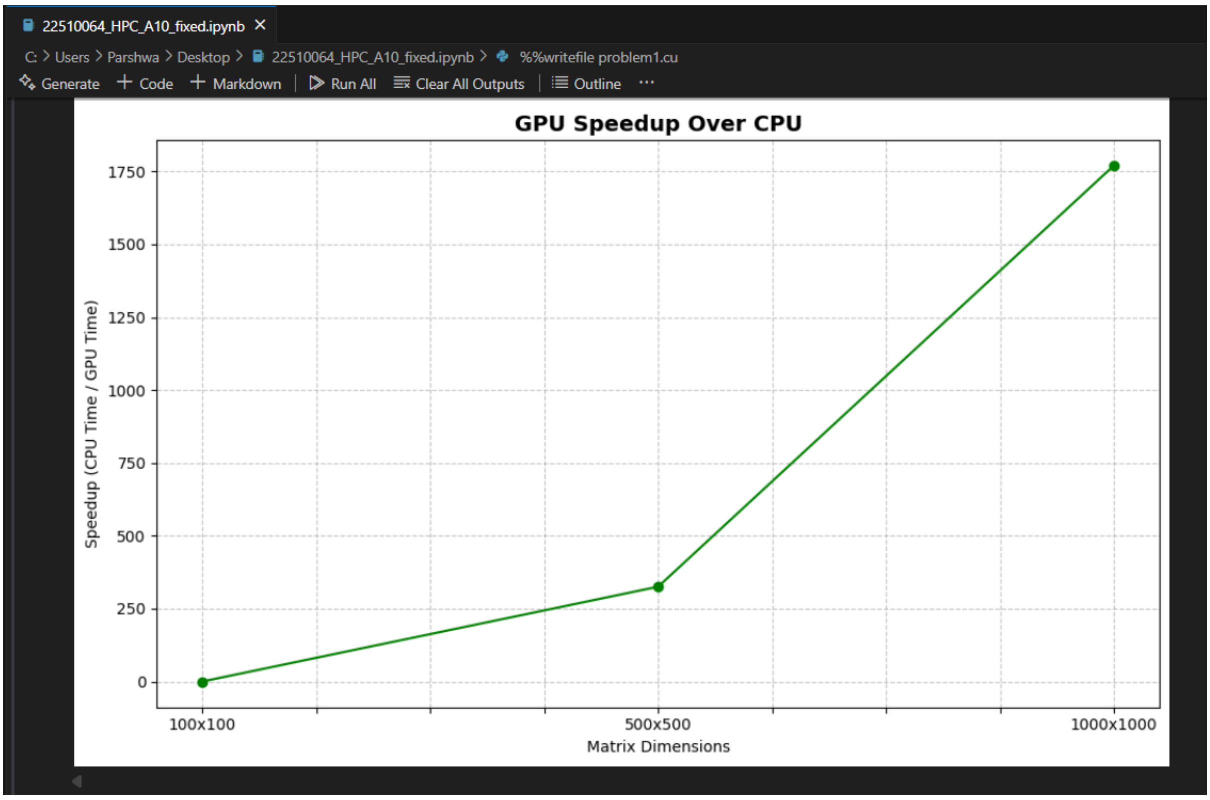
22510064_HPC_A10_fixed.ipynb ×

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All Clear All Outputs | Outline ···

```
Size,CPU_Time_ms,GPU_Time_ms
100x100,0.044093,7.50387
500x500,0.874233,0.002688
1000x1000,4.24869,0.0024

-------------------------

--- Matrix Addition Performance Analysis ---
        Size  CPU_Time_ms  GPU_Time_ms  Speedup
0    100x100         0.04          7.5     0.01
1    500x500         0.87          0.0   325.24
2  1000x1000         4.25          0.0  1770.29
```





Problem statement 7: Dot Product of Two Vectors using CUDA

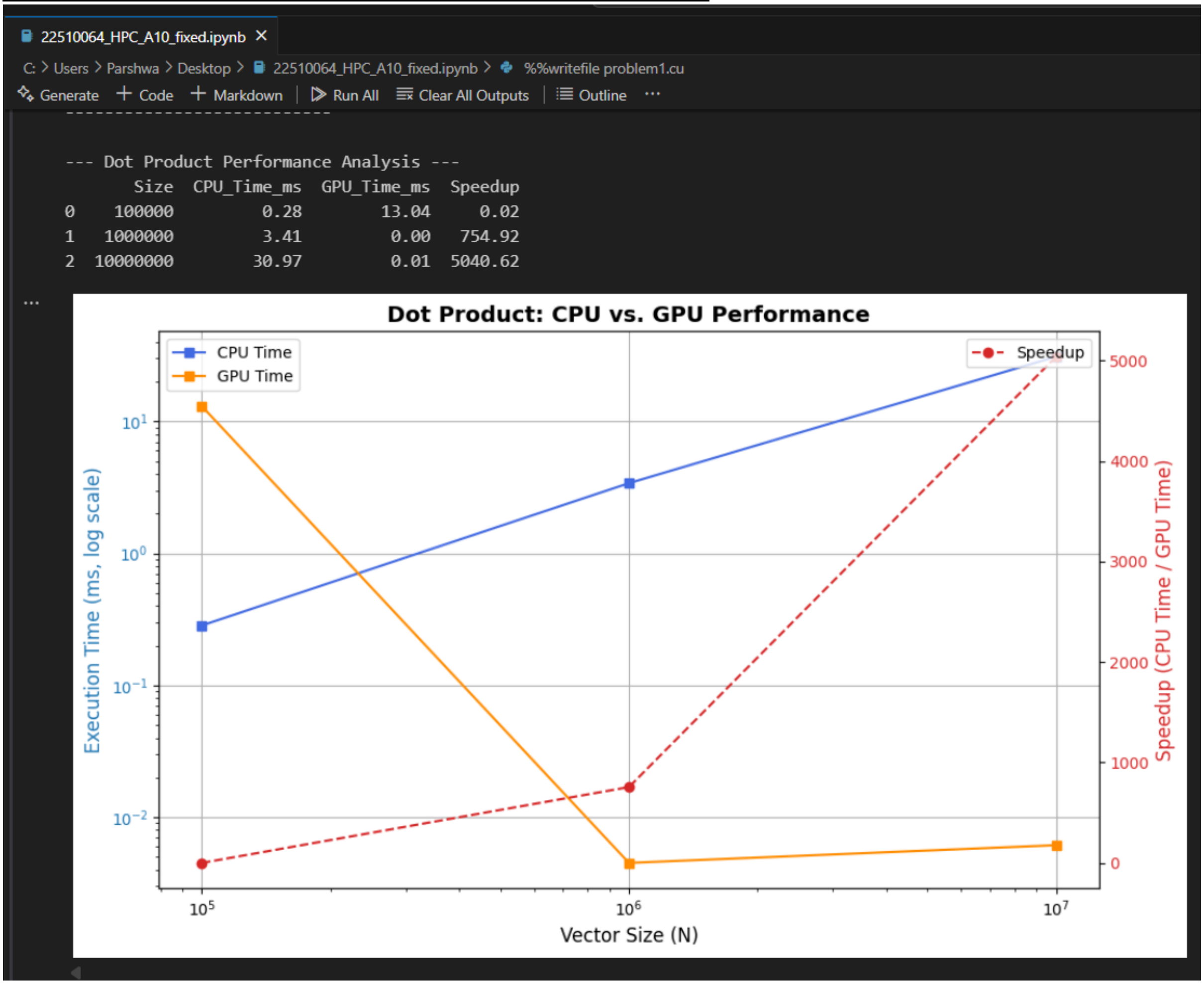
Problem Statement: Write a CUDA C program to compute the dot product of two vectors A and B of size N. The dot product is defined as:

Details:

- **Initialize the vectors A and B with random values.**
- **Implement the dot product calculation using both serial (CPU) and parallel (CUDA) approaches.**
- **Measure the execution time for both implementations with different vector sizes (e.g., N = 10^5, 10^6, 10^7).**
- **Use atomic operations or shared memory reduction in the CUDA kernel to compute the final sum.**

Task:

- **Calculate and report the speedup for different vector sizes.**



—

Problem statement 8: Matrix Multiplication using CUDA

Problem Statement: Write a CUDA C program to perform matrix multiplication. Given two matrices A (MxN) and B (NxP), compute the resulting matrix C (MxP) where:

Details:

- **Initialize the matrices A and B with random values.**
- **Write code for both serial (CPU-based) and parallel (CUDA-based) implementations.**
- **Measure the execution time of both implementations for various matrix sizes (e.g., 100x100, 500x500, 1000x1000).**

Task:

- **Calculate the speedup by comparing the CPU and GPU execution times.**

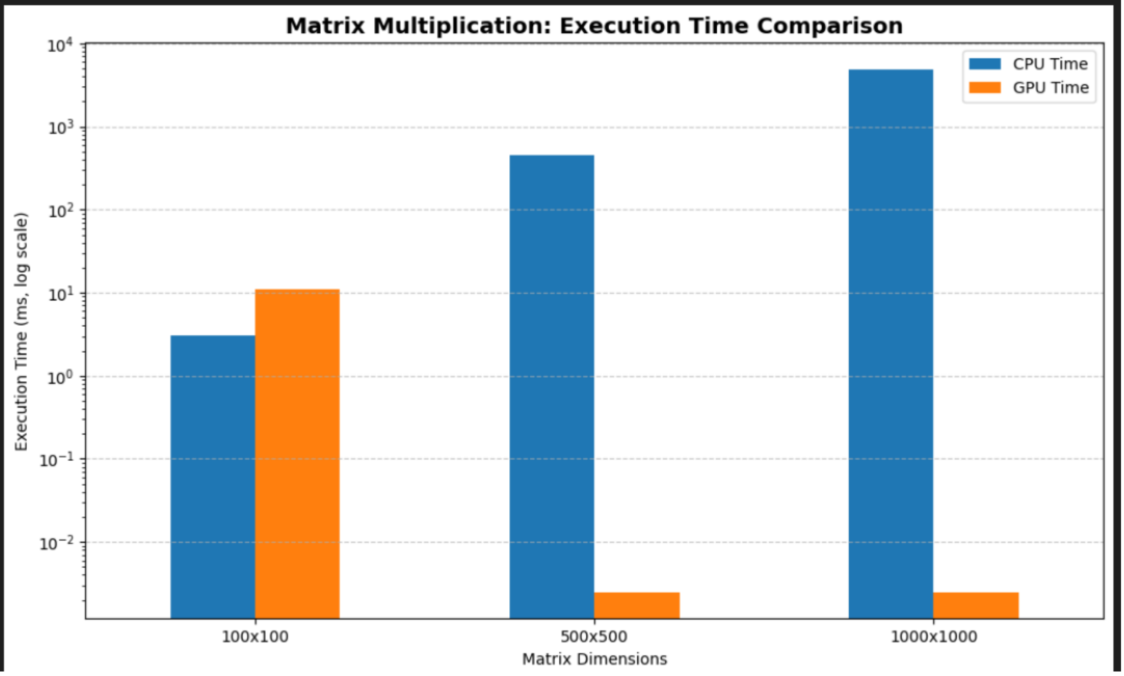
22510064_HPC_A10_fixed.ipynb ×

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All Clear All Outputs | Outline ...

```
--- Matrix Multiplication Performance Analysis ---
        Size  CPU_Time_ms  GPU_Time_ms      Speedup
0    100x100         3.08        10.98         0.28
1    500x500       457.42         0.00    185640.02
2  1000x1000      4936.23         0.00   2003340.10
```

...



22510064_HPC_A10_fixed.ipynb ×

C: > Users > Parshwa > Desktop > 22510064_HPC_A10_fixed.ipynb > %%writefile problem1.cu

Generate + Code + Markdown | Run All Clear All Outputs | Outline ...

Matrix Dimensions