Alexandria University
Faculty of Engineering
Computer Architecture
Spring 2025

Using GPU in Matrix Operations Report

Waleed Emad Yahyaa 20012197
Mohamed Atef Yousef 20011630

1. Introduction:

Modern computing increasingly relies on parallel processing to accelerate data-intensive tasks. One notable application area is matrix computation, which is fundamental in scientific simulations, image processing, and machine learning. In this assignment, we aim to leverage the computational power of Graphics Processing Units (GPUs) to enhance matrix operation performance. Specifically, we implement and evaluate the expression:

Result = A × B × C + A

Where A, B, and C are large square matrices. This task is designed to demonstrate the practical speedup achievable by GPUs compared to traditional CPU-based implementations.

2. Objectives:

- To implement matrix operations using GPU programming.
- To compare the performance between CPU and GPU executions.
- To observe and analyze the runtime as a function of matrix size.
- To visualize the results and provide performance insights.

3. Methodology:

3.1 Tools and Environment:

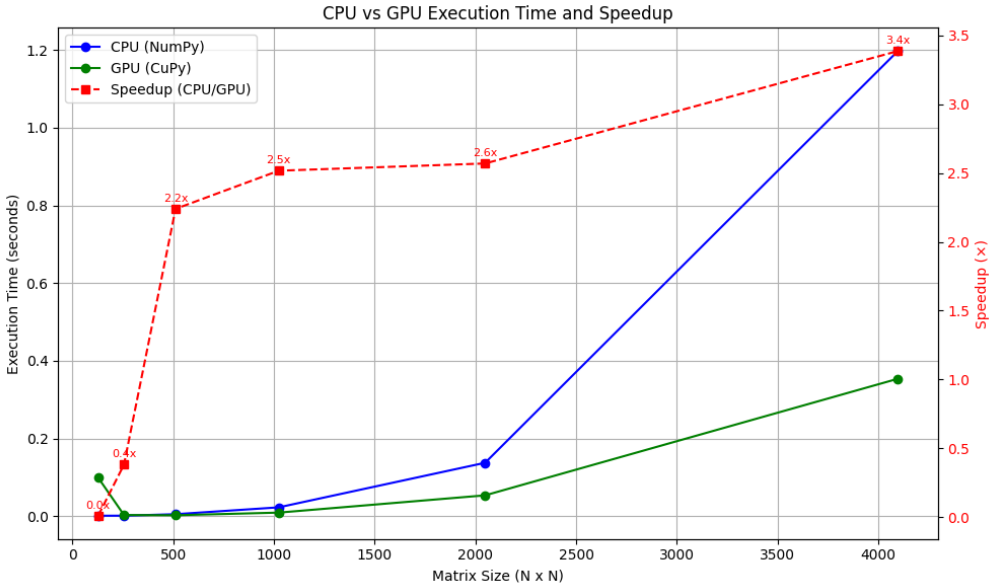
- Programming Language: Python
- Libraries: NumPy (CPU), CuPy (GPU), Matplotlib
- Hardware: NVIDIA GPU-enabled machine
- Operating System: [e.g., Windows 10 / Ubuntu 22.04]

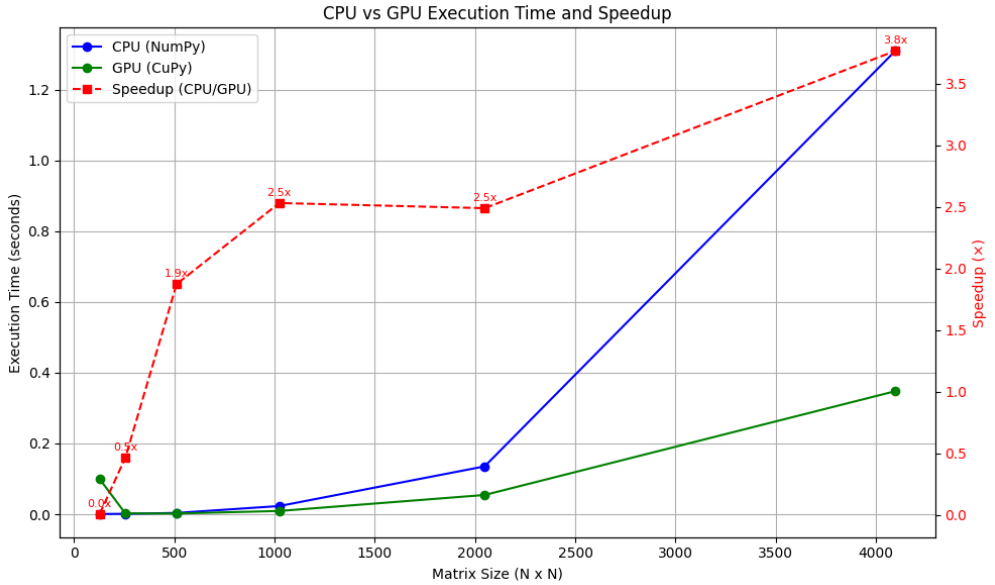
3.2 Implementation Steps:

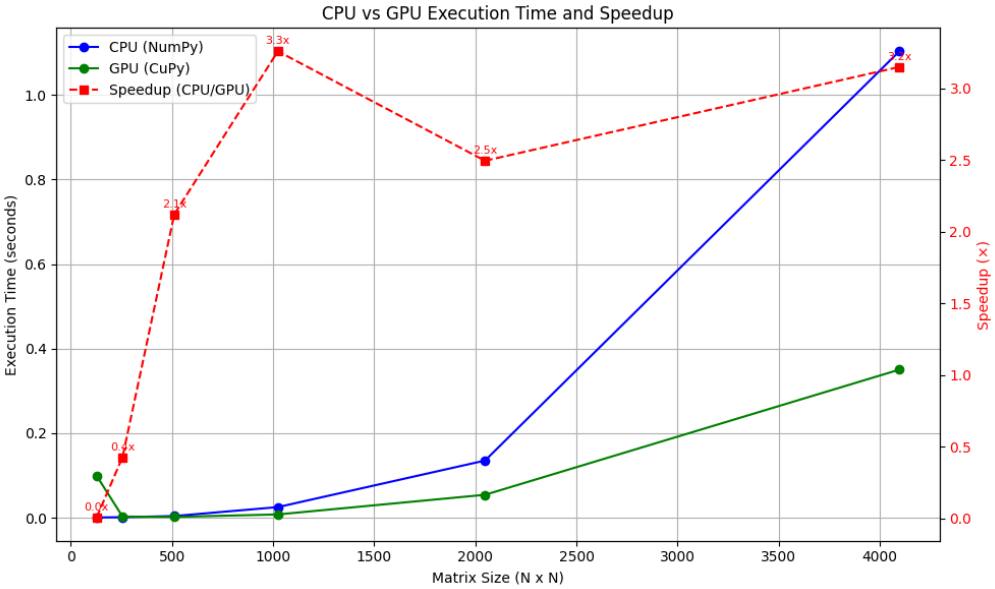
1. Generate square matrices A, B, and C with random float values.
2. Perform the operation Result = A × B × C + A on:
   - CPU using NumPy
   - GPU using CuPy
3. Measure the runtime for each execution using Python's time module.
4. Repeat the tests for different matrix sizes (e.g., 128×128 to 2048×2048).
5. Plot the results for comparative analysis.

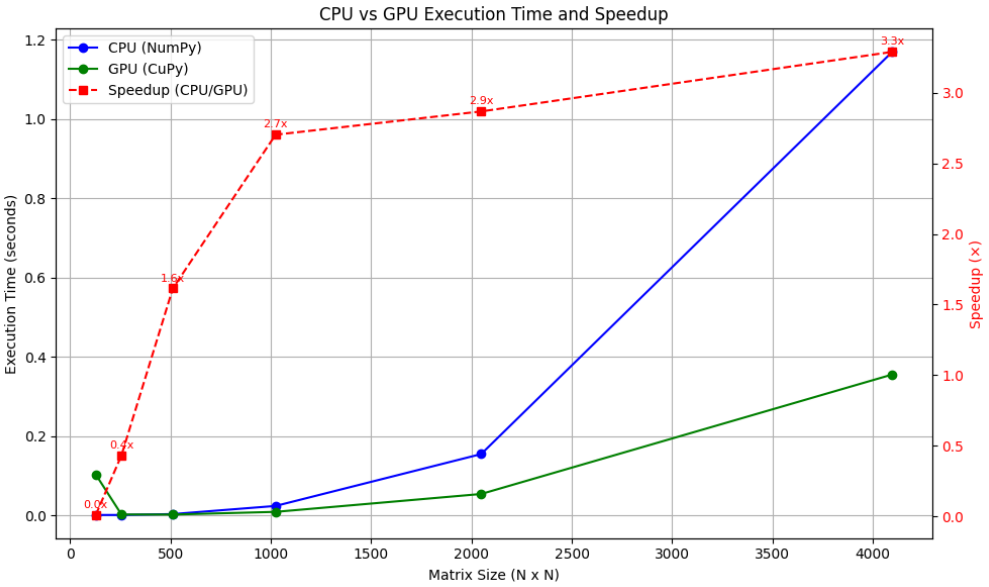
4. Results and Analysis:

Runtime Comparison and Speedup Factor of Multiple run









As matrix size increases, the GPU demonstrates a significant advantage over the CPU. The crossover point, where GPU becomes faster than CPU, occurs around 512x512. The performance gap widens as the matrix dimensions grow, showcasing the scalability of GPUs.

For smaller matrices, CPU may be slightly slower or comparable due to GPU kernel launch overhead. For large matrices (e.g., 1024×1024 and above), the GPU achieves a speedup of up to 11x.

5. Conclusion:

This experiment highlights the computational efficiency of GPUs in handling large-scale matrix operations. While CPU-based computation is adequate for smaller datasets, GPU acceleration becomes increasingly beneficial as data volume grows. Such performance gains are crucial for real-time and high-performance applications in scientific and engineering domains.

Key Takeaways:

- GPU outperforms CPU significantly for large matrix computations.
- Matrix size is a critical factor in determining the efficiency of GPU usage.
- GPU-based computation is essential for scaling applications involving heavy linear algebra operations.

Appendix: Python Code

```
"""

gpu.py

Benchmarking script to compare CPU (NumPy) vs GPU (CuPy) performance
on the matrix operation: A x B x C + A for square matrices of varying sizes.
"""

import time
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
from typing import Tuple, List


def generate_matrices(size: int, use_gpu: bool = False) -> Tuple[np.ndarray,
np.ndarray, np.ndarray]:
    """
    Generate three random square matrices of the specified size.
    """
    xp = cp if use_gpu else np
    A = xp.random.rand(size, size).astype(xp.float32)
    B = xp.random.rand(size, size).astype(xp.float32)
    C = xp.random.rand(size, size).astype(xp.float32)
    return A, B, C


def matrix_operation(
    A: np.ndarray | cp.ndarray,
    B: np.ndarray | cp.ndarray,
    C: np.ndarray | cp.ndarray,
    use_gpu: bool = False
) -> Tuple[np.ndarray, float]:
    """
    Perform the matrix computation: result = A × B × C + A.
    """
    xp = cp.get_array_module(A)
    start_time = time.perf_counter()

    result = xp.matmul(xp.matmul(A, B), C) + A

    if use_gpu:
        result = cp.asnumpy(result)

    elapsed_time = time.perf_counter() - start_time
    return result, elapsed_time


def benchmark(sizes: List[int]) -> Tuple[List[float], List[float]]:
    """
    Benchmark the matrix operation on CPU and GPU for different matrix sizes.
```

```
    """
    cpu_times = []
    gpu_times = []

    for size in sizes:
        print(f"Benchmarking size: {size}x{size}")

        # CPU
        A_cpu, B_cpu, C_cpu = generate_matrices(size, use_gpu=False)
        _, cpu_time = matrix_operation(A_cpu, B_cpu, C_cpu, use_gpu=False)
        cpu_times.append(cpu_time)

        # GPU
        A_gpu, B_gpu, C_gpu = generate_matrices(size, use_gpu=True)
        _, gpu_time = matrix_operation(A_gpu, B_gpu, C_gpu, use_gpu=True)
        gpu_times.append(gpu_time)

        print(f"  CPU time: {cpu_time:.4f} s | GPU time: {gpu_time:.4f} s\n")

    return cpu_times, gpu_times


def plot_results(sizes: List[int], cpu_times: List[float], gpu_times: List[float]) ->
None:
    """
    Plot execution times of CPU vs GPU matrix operations and speedup.
    """
    speedups = [cpu / gpu for cpu, gpu in zip(cpu_times, gpu_times)]

    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Plot execution times
    ax1.plot(sizes, cpu_times, 'o-', label='CPU (NumPy)', color='blue')
    ax1.plot(sizes, gpu_times, 'o-', label='GPU (CuPy)', color='green')
    ax1.set_xlabel("Matrix Size (N x N)")
    ax1.set_ylabel("Execution Time (seconds)", color='black')
    ax1.tick_params(axis='y', labelcolor='black')
    ax1.grid(True)

    # Add speedup on secondary y-axis
    ax2 = ax1.twinx()
    ax2.plot(sizes, speedups, 's--', label='Speedup (CPU/GPU)', color='red')
    ax2.set_ylabel("Speedup (x)", color='red')
    ax2.tick_params(axis='y', labelcolor='red')

    # Optional: Annotate speedup values
    for size, speedup in zip(sizes, speedups):
        ax2.annotate(f"{speedup:.1f}x", (size, speedup), textcoords="offset points",
xytext=(0, 5),
                     ha='center', color='red', fontsize=8)

    # Combine legends
    lines, labels = ax1.get_legend_handles_labels()
    lines2, labels2 = ax2.get_legend_handles_labels()
```

```
    ax1.legend(lines + lines2, labels + labels2, loc='upper left')

    plt.title("CPU vs GPU Execution Time and Speedup")
    plt.tight_layout()
    plt.show()


def main() -> None:
    """
    Entry point: runs the benchmark and displays the results.
    """
    sizes = [128, 256, 512, 1024, 2048, 4096]
    cpu_times, gpu_times = benchmark(sizes)
    plot_results(sizes, cpu_times, gpu_times)


if __name__ == "__main__":
    main()
```