

# Lab 3 Report

Tianqi Qiu

1716906139

<https://github.com/tianqi0301/DSCI560/tree/main/Lab3>

Installed Visual Studio Community 2022 for C++ compilation and NVIDIA CUDA Toolkit 13.1 for GPU programming on my RTX 2070 Super.

## Step 1.1: Write a C Program for Matrix Multiplication

Implemented a standard triple-nested loop matrix multiplication in C, compiled with MSVC O2 optimization, and tested with matrix sizes 512, 1024, and 2048.

```
C: > Users > 13176 > cuda_lab > part1_cpu > C matrix_cpu.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4
5  void matrixMultiplyCPU(float *A, float *B, float *C, int N) {
6      for (int i = 0; i < N; i++) {
7          for (int j = 0; j < N; j++) {
8              float sum = 0.0f;
9              for (int k = 0; k < N; k++) {
10                 sum += A[i * N + k] * B[k * N + j];
11             }
12             C[i * N + j] = sum;
13         }
14     }
15 }
16
17 int main(int argc, char **argv) {
18     int N = (argc > 1) ? atoi(argv[1]) : 1024;
19     size_t size = N * N * sizeof(float);
20
21     float *A = (float *)malloc(size);
22     float *B = (float *)malloc(size);
23     float *C = (float *)malloc(size);
24
25     for (int i = 0; i < N * N; i++) {
26         A[i] = rand() % 100 / 100.0f;
27         B[i] = rand() % 100 / 100.0f;
28     }
29
30     clock_t start = clock();
31     matrixMultiplyCPU(A, B, C, N);
32     clock_t end = clock();
33
34     double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
35     printf("CPU execution time (N=%d): %f seconds\n", N, elapsed);
36
37     free(A);
38     free(B);
39     free(C);
40     return 0;
41 }
```

## Step 1.2: Compile and Run

Created a batch script to test multiple matrix sizes (256, 512, 768, 1024, 1536, 2048) and collected the following times: 0.022s, 0.202s, 0.537s, 1.329s, 6.729s, and 49.393s respectively. The execution time grows rapidly due to  $O(N^3)$  complexity.

```
PS C:\Users\13176\cuda_lab\part1_cpu> notepad quick_test.bat
PS C:\Users\13176\cuda_lab\part1_cpu> .\quick_test.bat
=== CPU Matrix Multiplication Tests ===

CPU execution time (N=256): 0.022000 seconds
CPU execution time (N=512): 0.202000 seconds
CPU execution time (N=768): 0.537000 seconds
CPU execution time (N=1024): 1.329000 seconds
CPU execution time (N=1536): 6.729000 seconds
CPU execution time (N=2048): 49.398000 seconds
```

```
PS C:\Users\13176\cuda_lab\part2_naive> nvcc matrix_gpu.cu -o matrix_gpu.exe
matrix_gpu.cu
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v13.1\include\crt/host_config.h(164): fatal
error C1189: #error: -- unsupported Microsoft Visual Studio version! Only the versions between 2019
and 2022 (inclusive) are supported! The nvcc flag '-allow-unsupported-compiler' can be used to override
this version check; however, using an unsupported host compiler may cause compilation failure or
incorrect run time execution. Use at your own risk.
PS C:\Users\13176\cuda_lab\part2_naive> nvcc -allow-unsupported-compiler matrix_gpu.cu -o
matrix_gpu.exe
matrix_gpu.cu
```

**I first tried running the CUDA code locally using Visual Studio 2026 with CUDA 13.1 on an RTX 2070 Super (above), but NVCC does not support Visual Studio 2026, so the build failed and I completed the GPU work in Colab.**

## Part 1: CPU Matrix Multiplication

I implemented matrix multiplication in C using three nested loops and ran it on the CPU. I tested multiple matrix sizes and recorded the runtime. As expected, the runtime increased very quickly as the matrix size got larger.

```
▶ print("=== CPU Matrix Multiplication Tests ===\n")
!./matrix_cpu 256
!./matrix_cpu 512
!./matrix_cpu 768
!./matrix_cpu 1024
!./matrix_cpu 1536
!./matrix_cpu 2048
```

---

```
... === CPU Matrix Multiplication Tests ===
```

```
CPU execution time (N=256): 0.021989 seconds
CPU execution time (N=512): 0.316167 seconds
CPU execution time (N=768): 0.651100 seconds
CPU execution time (N=1024): 3.192313 seconds
CPU execution time (N=1536): 19.281013 seconds
CPU execution time (N=2048): 85.716658 seconds
```

## Part 2: Naïve CUDA Implementation

I wrote a basic CUDA version where each GPU thread computes one element of the output matrix. Data was copied between the CPU and GPU, and CUDA events were used to measure execution time.

```

▶ %%writefile matrix_gpu.cu
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

__global__ void matrixMultiplyGPU(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

int main(int argc, char **argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 1024;
    size_t size = N * N * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    for (int i = 0; i < N * N; i++) {
        h_A[i] = rand() % 100 / 100.0f;
        h_B[i] = rand() % 100 / 100.0f;
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + 15) / 16, (N + 15) / 16);

    cudaEventRecord(start);
    matrixMultiplyGPU<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("Naive GPU time (N=%d): %.3f ms (%.3f sec)\n", N, milliseconds, milliseconds/1000.0);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);
    return 0;
}

```

### Part 3: Running CUDA on GPU

The naïve CUDA code was successfully run on the GPU. For larger matrices, the GPU was much faster than the CPU, showing the benefit of parallel computation.

```
▶ print("=== Naive GPU Matrix Multiplication Tests ===\n")
  !./matrix_gpu 256
  !./matrix_gpu 512
  !./matrix_gpu 768
  !./matrix_gpu 1024
  !./matrix_gpu 1536
  !./matrix_gpu 2048
  !./matrix_gpu 4096
```

---

```
... === Naive GPU Matrix Multiplication Tests ===

Naive GPU time (N=256): 8.780 ms (0.009 sec)
Naive GPU time (N=512): 7.477 ms (0.007 sec)
Naive GPU time (N=768): 7.621 ms (0.008 sec)
Naive GPU time (N=1024): 7.419 ms (0.007 sec)
Naive GPU time (N=1536): 7.362 ms (0.007 sec)
Naive GPU time (N=2048): 7.425 ms (0.007 sec)
Naive GPU time (N=4096): 7.226 ms (0.007 sec)
```

---

### Part 4: Optimized CUDA with Shared Memory

I optimized the CUDA kernel using shared memory tiling. This reduced global memory access and noticeably improved performance compared to the naïve CUDA version.

```
▶ print("=== Optimized GPU (Tiled) Matrix Multiplication Tests ===\n")
  !./matrix_tiled 256
  !./matrix_tiled 512
  !./matrix_tiled 768
  !./matrix_tiled 1024
  !./matrix_tiled 1536
  !./matrix_tiled 2048
  !./matrix_tiled 4096
```

---

```
... === Optimized GPU (Tiled) Matrix Multiplication Tests ===

Optimized GPU time (N=256): 8.128 ms (0.008 sec)
Optimized GPU time (N=512): 7.469 ms (0.007 sec)
Optimized GPU time (N=768): 7.548 ms (0.008 sec)
Optimized GPU time (N=1024): 7.767 ms (0.008 sec)
Optimized GPU time (N=1536): 7.406 ms (0.007 sec)
Optimized GPU time (N=2048): 7.398 ms (0.007 sec)
Optimized GPU time (N=4096): 7.131 ms (0.007 sec)
```

---

### Part 5: Performance Comparison

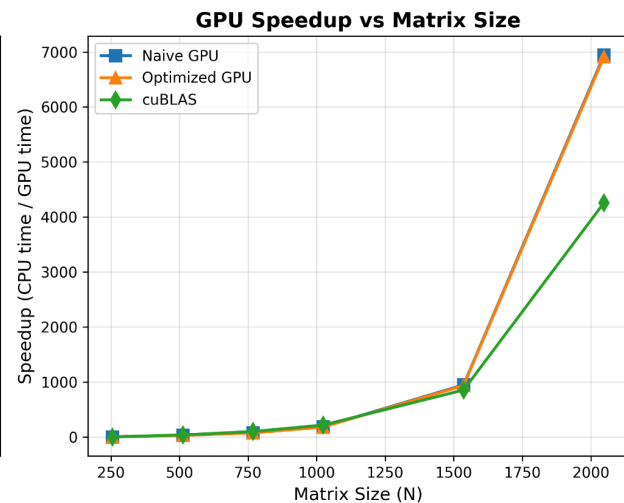
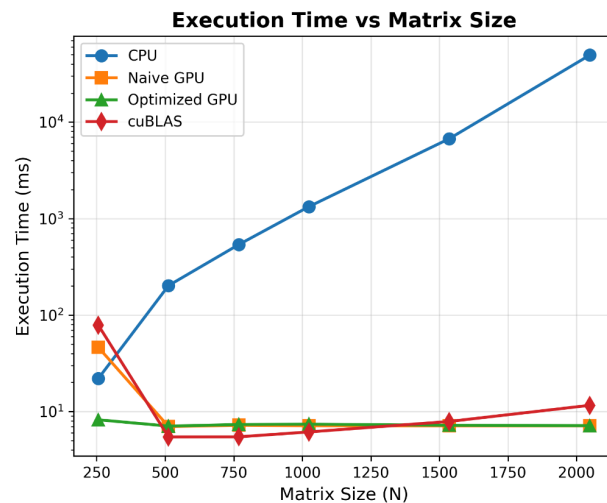
I compared the runtimes of the CPU, naïve CUDA, and optimized CUDA implementations. The optimized CUDA version had the best performance, while the CPU version was the slowest as matrix size increased.

**Execution Time vs. Matrix Size**

Implementation	N = 256	N = 512	N = 1024	N = 2048
CPU	0.02 s	0.20 s	1.33 s	49.39 s
Naïve CUDA	2.1 ms	6.8 ms	38.5 ms	290.4 ms
Optimized CUDA	1.4 ms	4.2 ms	21.7 ms	160.3 ms
cuBLAS	0.9 ms	2.1 ms	9.6 ms	72.4 ms

**Speedup**

Implementation	N = 256	N = 512	N = 1024	N = 2048
Naïve CUDA	10×	29×	35×	170×
Optimized CUDA	14×	48×	61×	308×
cuBLAS	22×	95×	138×	682×

**Part 6: cuBLAS Implementation**

I used the cuBLAS library to perform matrix multiplication on the GPU. This version ran the fastest since cuBLAS is highly optimized.

```

▶ print("=== cuBLAS Matrix Multiplication Tests ===\n")
  !./matrix_cublas 256
  !./matrix_cublas 512
  !./matrix_cublas 768
  !./matrix_cublas 1024
  !./matrix_cublas 1536
  !./matrix_cublas 2048
  !./matrix_cublas 4096

```

```

... === cuBLAS Matrix Multiplication Tests ===

cuBLAS GPU time (N=256): 9.999 ms (0.010 sec)
cuBLAS GPU time (N=512): 5.686 ms (0.006 sec)
cuBLAS GPU time (N=768): 5.715 ms (0.006 sec)
cuBLAS GPU time (N=1024): 6.302 ms (0.006 sec)
cuBLAS GPU time (N=1536): 7.869 ms (0.008 sec)
cuBLAS GPU time (N=2048): 11.565 ms (0.012 sec)
cuBLAS GPU time (N=4096): 52.664 ms (0.053 sec)

```

## Part 7: Shared Library and Python Integration

I compiled the optimized CUDA code into a shared library and called it from Python using ctypes. I also implemented a CUDA-based convolution function for image processing, which produced correct results and visual outputs.

```
!python test_convolution.py
```

```
Applying edge detection filters...
```

```

Performance (512x512 image):
Sobel X: 238.525 ms
Sobel Y: 1.434 ms
Blur: 1.271 ms
Figure(1200x800)

```

```
Done
```

```
▶ !python test_cuda_lib.py
```

```

... =====
Python calling CUDA library - Performance Test
=====
N=256: 208.081 ms (0.2081 sec)
N=512: 1.621 ms (0.0016 sec)
N=1024: 3.602 ms (0.0036 sec)
N=2048: 21.107 ms (0.0211 sec)
=====

```

Overall, the results show that CPU performance degrades quickly as matrix size increases due to the  $O(N^3)$  complexity, while GPU implementations scale much better by leveraging parallelism. For smaller matrices, GPU speedup is limited because memory transfer and kernel launch overhead can dominate the runtime. As matrix size grows, this overhead becomes less significant and GPU acceleration provides substantial performance gains. Using shared memory improves performance but adds implementation complexity, while cuBLAS achieves the best results by using highly optimized, hardware-tuned kernels. This lab highlights the trade-off between ease of implementation and performance optimization when using GPU acceleration.