EE 533 Spring 2026

CUDA Laboratory : CUDA Program and Custom Python Library

Student Name: Yizheng Qiao
USC ID: 8026907748

Part 1.

The goal of the first part of this experiment was to implement matrix multiplication in C on a CPU and test and analyze its performance. The experiment involved writing a matrix multiplication program based on a triple loop to calculate the product of two N×N floating-point matrices. The program supports specifying the matrix size via command-line arguments, allowing for testing performance with different matrix dimensions.

During program implementation, memory space for the matrices was dynamically allocated, and the input matrices were initialized with randomly generated floating-point numbers. The matrix multiplication function was then called to perform the calculation, and system-provided timing functions were used to record the start and end times of the calculation, thus obtaining the execution time of matrix multiplication on the CPU.

```
C:\Users\11148\Desktop\Cuda_LAB>cl matrix_cpu.c
用于 x64 的 Microsoft (R) C/C++ 优化编译器 19.50.35723 版
版权所有(C) Microsoft Corporation。保留所有权利。

matrix_cpu.c
Microsoft (R) Incremental Linker Version 14.50.35723.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:matrix_cpu.exe
matrix_cpu.obj

C:\Users\11148\Desktop\Cuda_LAB>matrix_cpu 128
CPU execution time (N=128): 0.004000 seconds

C:\Users\11148\Desktop\Cuda_LAB>matrix_cpu 256
CPU execution time (N=256): 0.033000 seconds

C:\Users\11148\Desktop\Cuda_LAB>matrix_cpu 512
CPU execution time (N=512): 0.264000 seconds

C:\Users\11148\Desktop\Cuda_LAB>matrix_cpu 1024
CPU execution time (N=1024): 2.244000 seconds

C:\Users\11148\Desktop\Cuda_LAB>matrix_cpu 2048
CPU execution time (N=2048): 37.950000 seconds

C:\Users\11148\Desktop\Cuda_LAB>
```

By running the program multiple times with different matrix sizes, such as N=128, 512, 1024, and 2048, and recording the corresponding execution times, we can organize and plot the experimental data to visually observe the impact of increasing matrix size on CPU computing performance and execution time.

Part 2.

The second part of the experiment mainly focuses on learning the basic usage of CUDA programming and attempting to implement matrix multiplication on the GPU. A simple CUDA kernel was written for this experiment, where each GPU thread is responsible for calculating one element of the resulting matrix. The thread and thread block indices are used to determine the corresponding matrix position for the current thread, and the matrix multiplication is performed only when the boundary conditions are met.

```
C:\Users\11148\Desktop\Cuda_LAB>nvcc -allow-unsupported-compiler matrix_gpu.cu -o matrix_gpu
matrix_gpu.cu
tmpxft_0000a33c_00000000-7_matrix_gpu.cudafe1.cpp

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu 512
Na├»ve CUDA execution time (N=512): 15.861 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu 1024
Na├»ve CUDA execution time (N=1024): 16.653 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu 2048
Na├»ve CUDA execution time (N=2048): 24.387 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu 4096
Na├»ve CUDA execution time (N=4096): 90.413 ms
```

After completing the program implementation, the CUDA program was run with matrices of different sizes, such as 512, 1024, and 2048, and the program's execution time was measured using the same method as in the first part of the experiment, in order to observe the GPU's performance in matrix multiplication.

Part 3.

Part 4.

```
C:\Users\11148\Desktop\Cuda_LAB>nvcc -allow-unsupported-compiler matrix_gpu_opt.cu -o matrix_gpu_opt
matrix_gpu_opt.cu
tmpxft_00005a9c_00000000-7_matrix_gpu_opt.cudafe1.cpp

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu_opt 512
Optimized CUDA (Tiled) execution time (N=512): 15.724 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu_opt 512
Optimized CUDA (Tiled) execution time (N=512): 14.684 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu_opt 1024
Optimized CUDA (Tiled) execution time (N=1024): 15.674 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu_opt 2048
Optimized CUDA (Tiled) execution time (N=2048): 21.530 ms

C:\Users\11148\Desktop\Cuda_LAB>matrix_gpu_opt 4096
Optimized CUDA (Tiled) execution time (N=4096): 69.243 ms

C:\Users\11148\Desktop\Cuda_LAB>
```

The fourth part of the experiment focuses on optimizing the CUDA matrix multiplication program, primarily using shared memory and tiling techniques to improve performance. In this implementation, the matrices are divided into smaller blocks of size TILE_WIDTH×TILE_WIDTH. Each thread block first loads the

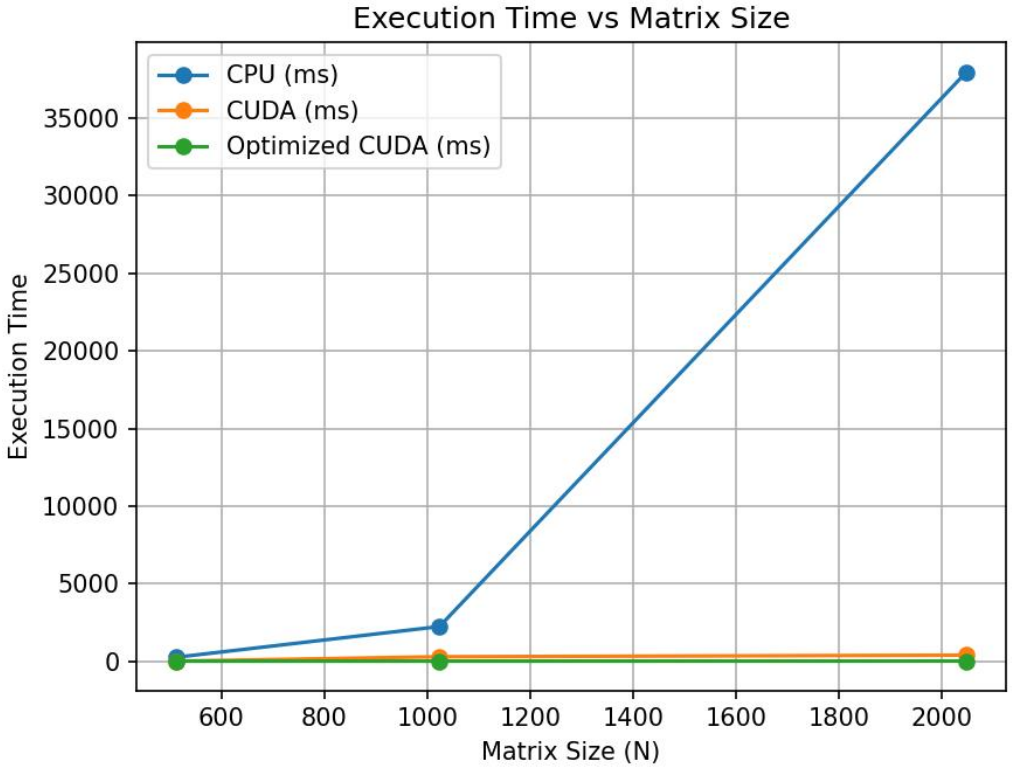
required data from global memory into shared memory before performing calculations, thus reducing repeated access to global memory.

Within the kernel, each thread is responsible for loading a portion of matrices A and B into shared memory, and thread synchronization ensures that the data is fully loaded before computation begins. This approach improves memory access efficiency, thereby accelerating the execution speed of matrix multiplication. The optimized CUDA program was run with matrix sizes of 512, 1024, and 2048, and the corresponding execution times were recorded for comparison with the performance of the un- optimized version.

Part 5.

The table below shows the execution times for the CPU, unoptimized CUDA, and optimized CUDA implementations at different matrix sizes. CPU times are given in seconds (s), and GPU times are given in milliseconds (ms).

	N = 512	N = 1024	N = 2048
CPU	0.264 s	2.244 s	37.950 s
CUDA	15.861 ms	16.653 ms	24.387 ms
Optimized CUDA（Tiled）	14.684 ms	15.674 ms	21.530 ms



Speedup is defined as the ratio of CPU execution time to GPU execution time: Speedup = CPU time / GPU time

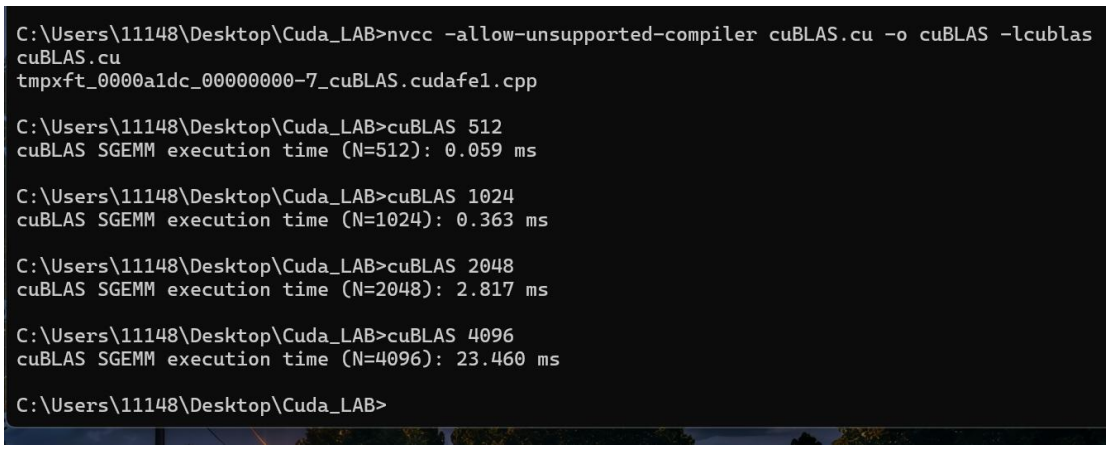
	N = 512	N = 1024	N = 2048
CUDA Speedup	16.6 X	134.7 X	1556 X
Optimized CUDA Speedup	18.0 X	143.2 X	1763 X

The experimental results show that as the matrix size increases, the execution time of matrix multiplication on the CPU increases rapidly, while the execution time of the GPU implementation increases relatively slowly. This indicates that the GPU has a significant performance advantage in handling large-scale parallel computing tasks.

For the CUDA implementation, although it utilizes the parallel computing capabilities of the GPU, its performance is still somewhat limited due to frequent access to global memory. By introducing shared memory and tiling, the optimized CUDA implementation further reduces the number of global memory accesses, thus achieving better performance across all matrix sizes.

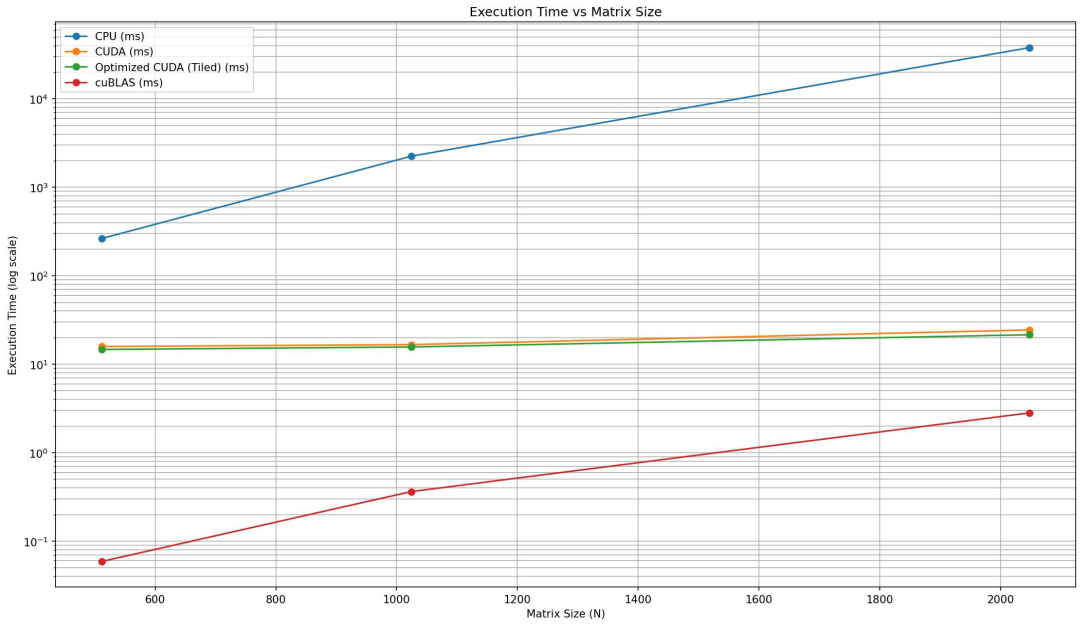
Part 6.

This section implements matrix multiplication b using the cuBLAS library included in the CUDA Toolkit. Single-precision matrix multiplication is performed on the GPU by calling the cublasSgemm function. Here, the matrices are not transposed, N is the dimension of the matrices, alpha and beta are the multiplication and accumulation coefficients respectively, and d_A, d_B, and d_C are pointers to the matrices on the device.



	N = 512	N = 1024	N = 2048
CPU	0.264 s	2.244 s	37.950 s

CUDA	15.861 ms	16.653 ms	24.387 ms
Optimized CUDA（Tiled）	14.684 ms	15.674 ms	21.530 ms
cuBLAS	0.059ms	0.363ms	2.817ms



Part 7.

Analysis Questions

1. How does performance change as matrix size increases?

As the matrix size increases, the CPU execution time grows very rapidly, exhibiting a clear non-linear growth trend. In contrast, the GPU execution time increases relatively slowly, especially with optimized CUDA and cuBLAS, which maintain high computational efficiency even with large matrix sizes. This demonstrates that GPUs are better suited for handling large-scale parallel computing tasks.

2. At what point does the GPU significantly outperform the CPU?

When the matrix size is large and the computational load is sufficiently high, the parallel computing capabilities of the GPU can be fully utilized, resulting in significantly better performance than the CPU. For smaller problems, the data transfer and startup overhead of the GPU may offset some of the performance advantages, but these overheads are relatively small in large-scale matrix operations.

3. How much speedup is gained by tiling optimization vs. naïve CUDA?

Compared to the basic CUDA implementation, the execution time was significantly reduced after applying tiling optimization, resulting in a further improvement in overall performance. Tiling optimization effectively improves computational efficiency by reducing the number of accesses to global memory and

increasing the utilization of shared memory, and its effect is particularly noticeable with larger matrix sizes.

4. How close is your optimized kernel to cuBLAS performance?

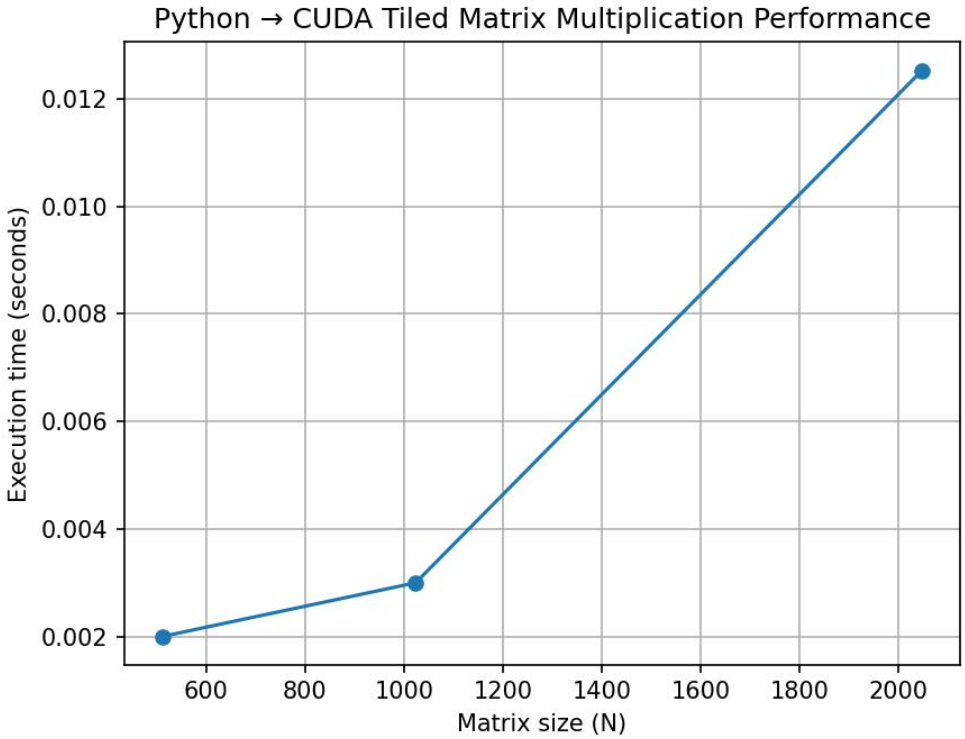
The experimental results show that although the optimized CUDA kernel performs significantly better than the naive implementation, its performance is still lower than cuBLAS. The difference between the two becomes more pronounced as the matrix size increases, with cuBLAS demonstrating higher execution efficiency across all scales.

5. Why might cuBLAS still outperform hand-written kernels?

cuBLAS is a highly optimized mathematical library developed by NVIDIA, deeply optimized for different GPU architectures. It employs more complex and efficient algorithms and fully utilizes hardware capabilities. In contrast, manually written kernels are still limited in terms of optimization and adaptability, and therefore generally do not perform as well as cuBLAS.

I need to compile the CUDA matrix multiplication kernel implemented in Part 4 into a shared library and modify the CUDA code to provide an external interface function so that it can be called via Python's ctypes, allowing me to use GPU-accelerated matrix multiplication in my Python program.

```
C:\Users\11148\Desktop\Cuda_LAB>python test_matrix.py
Python → CUDA tiled matrix multiplication
----------------------------------------
N = 512  | Time = 0.0020 seconds
N = 1024 | Time = 0.0030 seconds
N = 2048 | Time = 0.0125 seconds
----------------------------------------
```



Step 1.

This section primarily focuses on the implementation and testing of a two-dimensional image convolution function based on the C language: a convolution function was implemented to apply an N×N filter to an M×M grayscale image; various image processing filters were designed and used; multiple sample images were selected and processed; the correctness of the convolution function was verified through the output images; and performance data was collected and recorded under different combinations of filter size N and image size M.

```
conv_cpu.c
Microsoft (R) Incremental Linker Version 14.50.35723.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:conv_cpu.exe
conv_cpu.obj

C:\Users\11148\Desktop\Cuda_LAB>conv_cpu.exe
M=512 N=3 filter=blur3x3 time=0.0020 sec
M=512 N=3 filter=sobelx time=0.0010 sec
M=512 N=3 filter=laplacian time=0.0010 sec
M=512 N=5 filter=blur5x5 time=0.0050 sec
M=512 N=7 filter=blur7x7 time=0.0090 sec
M=1024 N=3 filter=blur3x3 time=0.0090 sec
M=1024 N=3 filter=sobelx time=0.0030 sec
M=1024 N=3 filter=laplacian time=0.0030 sec
M=1024 N=5 filter=blur5x5 time=0.0200 sec
M=1024 N=7 filter=blur7x7 time=0.0370 sec
M=2048 N=3 filter=blur3x3 time=0.0340 sec
M=2048 N=3 filter=sobelx time=0.0120 sec
M=2048 N=3 filter=laplacian time=0.0120 sec
M=2048 N=5 filter=blur5x5 time=0.0790 sec
M=2048 N=7 filter=blur7x7 time=0.1490 sec
All done.

C:\Users\11148\Desktop\Cuda_LAB>
```

Step 2.

I need to rewrite the image convolution function previously implemented in C language into a CUDA version, perform the same convolution operation on the GPU, and then test the execution time with the same image and filter sizes. Finally, I need to compare and analyze the performance of the GPU version against the CPU version.

```
tmpxft_00005088_00000000-7_conv_compare.cudafe1.cpp

C:\Users\11148\Desktop\Cuda_LAB>./conv_compare
'.' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\11148\Desktop\Cuda_LAB>conv_compare.exe
M,N,filter,cpu_sec,gpu_total_sec,gpu_kernel_ms
512,3,blur3x3,0.002000,0.020000,0.017
512,5,blur5x5,0.005000,0.001000,0.020
512,7,blur7x7,0.009000,0.000000,0.027
512,3,sobelx,0.000000,0.000000,0.031
512,3,laplacian,0.001000,0.000000,0.017
1024,3,blur3x3,0.009000,0.002000,0.201
1024,5,blur5x5,0.021000,0.001000,0.042
1024,7,blur7x7,0.038000,0.001000,0.089
1024,3,sobelx,0.003000,0.001000,0.027
1024,3,laplacian,0.004000,0.001000,0.028
2048,3,blur3x3,0.034000,0.006000,0.119
2048,5,blur5x5,0.080000,0.004000,0.180
2048,7,blur7x7,0.154000,0.004000,0.264
2048,3,sobelx,0.012000,0.004000,0.118
2048,3,laplacian,0.011000,0.004000,0.122
All done.

C:\Users\11148\Desktop\Cuda_LAB>
```

Next,I integrated the already implemented CUDA - accelerated convolution functions into a custom shared library, and then called these functions from Python using this library to achieve accelerated computation. I then compared and analyzed the performance of the Python - CUDA implementation against both a pure C version and a directly executed CUDA program.

```
C:\Users\11148\Desktop\Cuda_LAB>python quick_test_dll.py
call succeeded, out[0] = 27

C:\Users\11148\Desktop\Cuda_LAB>conv_cpu.exe > cpu_out.txt

C:\Users\11148\Desktop\Cuda_LAB>conv_compare.exe > conv_compare_out.txt

C:\Users\11148\Desktop\Cuda_LAB>python test_conv_library.py
[GPU DLL] M=512 N=3 time=0.001011 s
[CPU EXE]  M=512 N=3 time=0.003000 s
[CUDA EXE] M=512 N=3 time=0.024000 s
[GPU DLL] M=512 N=5 time=0.002016 s
[CPU EXE]  M=512 N=5 time=0.006000 s
[CUDA EXE] M=512 N=5 time=0.001000 s
[GPU DLL] M=512 N=7 time=0.001750 s
[CPU EXE]  M=512 N=7 time=0.010000 s
[CUDA EXE] M=512 N=7 time=0.002000 s
[GPU DLL] M=1024 N=3 time=0.002995 s
[CPU EXE]  M=1024 N=3 time=0.009000 s
[CUDA EXE] M=1024 N=3 time=0.002000 s
[GPU DLL] M=1024 N=5 time=0.002999 s
[CPU EXE]  M=1024 N=5 time=0.021000 s
[CUDA EXE] M=1024 N=5 time=0.002000 s
[GPU DLL] M=1024 N=7 time=0.002141 s
[CPU EXE]  M=1024 N=7 time=0.043000 s
[CUDA EXE] M=1024 N=7 time=0.003000 s
[GPU DLL] M=2048 N=3 time=0.005519 s
[CPU EXE]  M=2048 N=3 time=0.043000 s
[CUDA EXE] M=2048 N=3 time=0.007000 s
[GPU DLL] M=2048 N=5 time=0.006023 s
[CPU EXE]  M=2048 N=5 time=0.091000 s
[CUDA EXE] M=2048 N=5 time=0.006000 s
[GPU DLL] M=2048 N=7 time=0.005989 s
[CPU EXE]  M=2048 N=7 time=0.167000 s
[CUDA EXE] M=2048 N=7 time=0.004000 s
```