EE533 CUDA Lab

Project GitHub Link: https://github.com/jeremycai99/EE533-Labs/tree/main/CUDA

Introduction

In this lab, we explore how GPU acceleration changes the performance of compute-intensive workloads by implementing and benchmarking matrix multiplication across a progression of approaches: a baseline CPU implementation, a naive CUDA kernel, an optimized CUDA kernel using shared-memory tiling, and NVIDIA's cuBLAS library routine. By measuring execution time across multiple problem sizes, we analyze performance scaling, quantify speedups, and examine the overheads involved in offloading work to an accelerator. The lab also demonstrates practical deployment on a GPU-enabled cloud VM and extends beyond standalone programs by compiling CUDA code into a shared library that can be called from Python using ctypes, highlighting how GPU acceleration can be integrated into higher-level workflows.

The setup for this Lab is Windows 11 PC with NVIDIA RTX3070 laptop, CUDA version 13.1 under driver firmware 591.74.

Please note that for Part 3, which requires using Google Compute Engineer as naive CUDA kernel platform, hasn't been completed in this Lab. Due to cloud resource restrictions, we are not able to allocate the cloud GPU resource for completing this part. Please see more details in corresponding session.

Part 1: Matrix Multiplication on the CPU

In this part, we implement a simple C code in order to run matrix multiplication on CPU as benchmark for GPU speedup exploration. Since code is provided in the lab manual, we won't attach it here.

We implement the testing scheme to meet following requirements:

1. Use same Ns: 256, 512, 1024, 2048, 4096
2. Run 5 times for each N in each test and use average as performance representative
3. All measurement based on execution time (meaning it may fluctuate if CPU/GPU load is varying

The script file used for CPU matrix multiplication on Windows:

```
@echo off
setlocal enabledelayedexpansion
set SRC=matrix_cpu.c
set EXE=matrix_cpu.exe
set CSV=matrix_cpu_results.csv
set TRIALS=5
echo Compiling...
gcc %SRC% -O2 -o %EXE%
if errorlevel 1 (
  echo Compile failed.
  exit /b 1
)
echo timestamp,N,trial,exit_code,stdout > %CSV%
for %%N in (256 512 1024 2048) do (
  echo Running N=%%N...
  for /L %%T in (1,1,%TRIALS%) do (
    for /f "usebackq delims=" %%O in (`powershell -NoProfile -Command "Get-Date -Forma
    for /f "usebackq delims=" %%R in (`%EXE% %%N 2^>^&1`) do (
```

```
      set OUT=%%R
      echo "!TS!",%%N,%%T,0,"!OUT!" >> %CSV%
    )
  )
)
echo Done. Results saved to %CSV%
endlocal
```

We conclude the following latency/performance raw data (in ms) into following table:

Table 1: Runtime Summary for CPU Based Matrix Multiplication

	Run #\N	256	512	1024	2048	4096
1	Run 1	13	146	2620	39410	654770
2	Run 2	16	179	2570	30236	710714
3	Run 3	15	151	2526	31705	642368
4	Run 4	15	135	2556	32956	638417
5	Run 5	15	144	2651	20793	633106
6	Average	14.8	151	2584.6	31020	655875

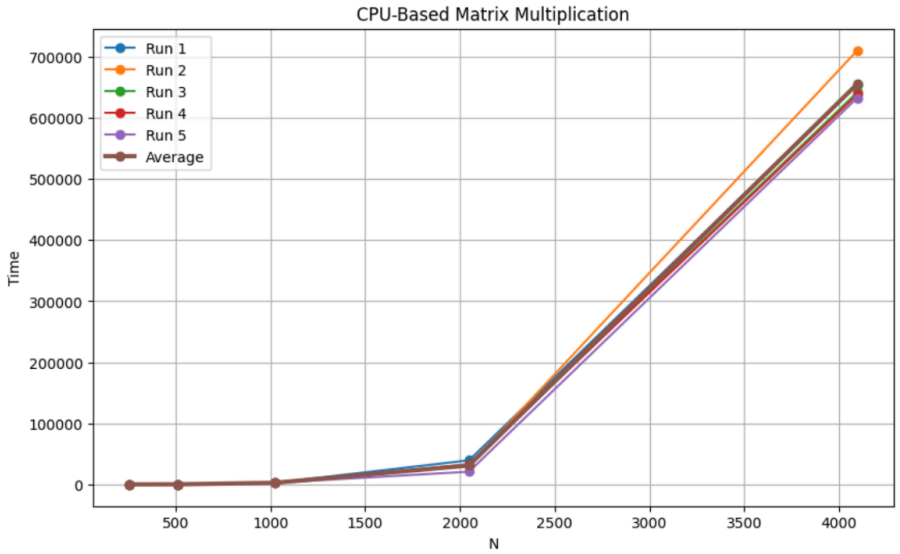

Figure 1. Line Graph for CPU Based Matrix Multiplication

Part 2: Naive CUDA Kernel Design

In this part, we implement CUDA kernel for the same matrix multiplication as in Part 1. Attach the full code as follows:

```
#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 = 1024;
    if (argc > 1) N = atoi(argv[1]);

    size_t bytes = (size_t)N * (size_t)N * sizeof(float);

    // Host memory
    float *hA = (float*)malloc(bytes);
    float *hB = (float*)malloc(bytes);
    float *hC = (float*)malloc(bytes);

    if (!hA || !hB || !hC) {
        printf("Host malloc failed\n");
        return 1;
    }

    // Initialize A and B
    for (int i = 0; i < N * N; i++) {
        hA[i] = 1.0f;
        hB[i] = 1.0f;
    }

    // Device memory
    float *dA = NULL, *dB = NULL, *dC = NULL;
    if (cudaMalloc((void**)&dA, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dB, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dC, bytes) != cudaSuccess) {
        printf("cudaMalloc failed\n");
        return 1;
    }

    // Copy to device
    if (cudaMemcpy(dA, hA, bytes, cudaMemcpyHostToDevice) != cudaSuccess ||
        cudaMemcpy(dB, hB, bytes, cudaMemcpyHostToDevice) != cudaSuccess) {
        printf("cudaMemcpy H2D failed\n");
        return 1;
    }

    // Launch config
    dim3 block(16, 16);
    dim3 grid((N + block.x - 1) / block.x, (N + block.y - 1) / block.y);

    // ---- Timing: CUDA events (kernel time only) ----
```

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

    // Optional warmup (keeps timing more stable)
    matrixMultiplyGPU<<<grid, block>>>(dA, dB, dC, N);
    cudaDeviceSynchronize();

    cudaEventRecord(start);
    matrixMultiplyGPU<<<grid, block>>>(dA, dB, dC, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms = 0.0f;
    cudaEventElapsedTime(&ms, start, stop);

    // Minimal error check
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Kernel launch error: %s\n", cudaGetErrorString(err));
        return 1;
    }

    // Copy result back (also syncs)
    if (cudaMemcpy(hC, dC, bytes, cudaMemcpyDeviceToHost) != cudaSuccess) {
        printf("cudaMemcpy D2H failed\n");
        return 1;
    }

    // Output in a CPU-like way: report size + time
    printf("GPU (naive cuda) execution time N=%d, time_ms=%f, C0=%f\n", N, ms, hC[0]);

    // Cleanup
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    free(hA);
    free(hB);
    free(hC);

    return 0;
}
```

We therefore use following modified script for Part 2:

```
@echo off
setlocal enabledelayedexpansion

set SRC=matrix_naive_cuda.cu
set EXE=matrix_naive_cuda.exe
set CSV=matrix_naive_cuda_results.csv
```

```
set TRIALS=5

echo Compiling...
nvcc %SRC% -o %EXE%
if errorlevel 1 (
  echo Compile failed.
  exit /b 1
)

echo timestamp,N,trial,exit_code,stdout > %CSV%

for %%N in (256 512 1024 2048 4096) do (
  echo Running N=%%N...
  for /L %%T in (1,1,%TRIALS%) do (
    for /f "usebackq delims=" %%O in (`powershell -NoProfile -Command "Get-Date -Forma

    REM Capture program output (works best if program prints ONE line)
    set OUT=
    for /f "usebackq delims=" %%R in (`.\%EXE% %%N 2^>^&1`) do set OUT=%%R
    set EXITCODE=!errorlevel!

    echo "!TS!",%%N,%%T,!EXITCODE!,"!OUT!" >> %CSV%
  )
)

echo Done. Results saved to %CSV%
endlocal
```

Similarly, we obtain the raw data:

Table 2. Runtime Summary for Naive CUDA Based Matrix Multiplication

	Run #\N	256	512	1024	2048	4096
1	Run 1	0.045	0.275	2.127	16.831	142.482
2	Run 2	0.046	0.278	2.126	18.169	139.167
3	Run 3	0.046	0.278	2.491	18.222	141.134
4	Run 4	0.046	0.278	2.127	17.31	142.275
5	Run 5	0.046	0.279	2.128	16.829	140.896
6	Average	0.0458	0.2776	2.1998	17.4722	141.1908

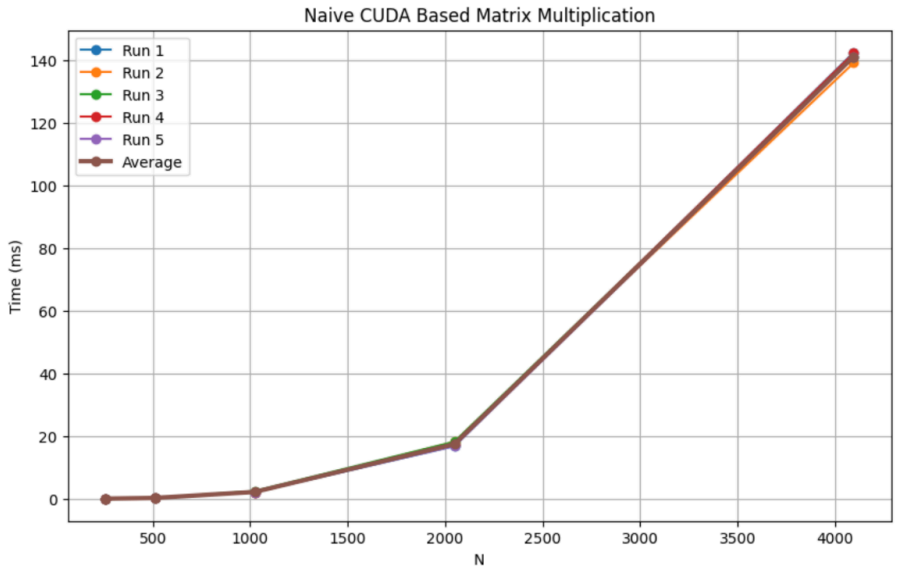

Figure 2. Line Graph for Naive CUDA Based Matrix Multiplication

Part 3: Running CUDA on Google Cloud (Skipped)

As described in report introduction, due to cloud resource limitation, we are not able to allocate cloud GPU resource for this part. Also, we tried to apply for quota but not approved at time we draft this report.

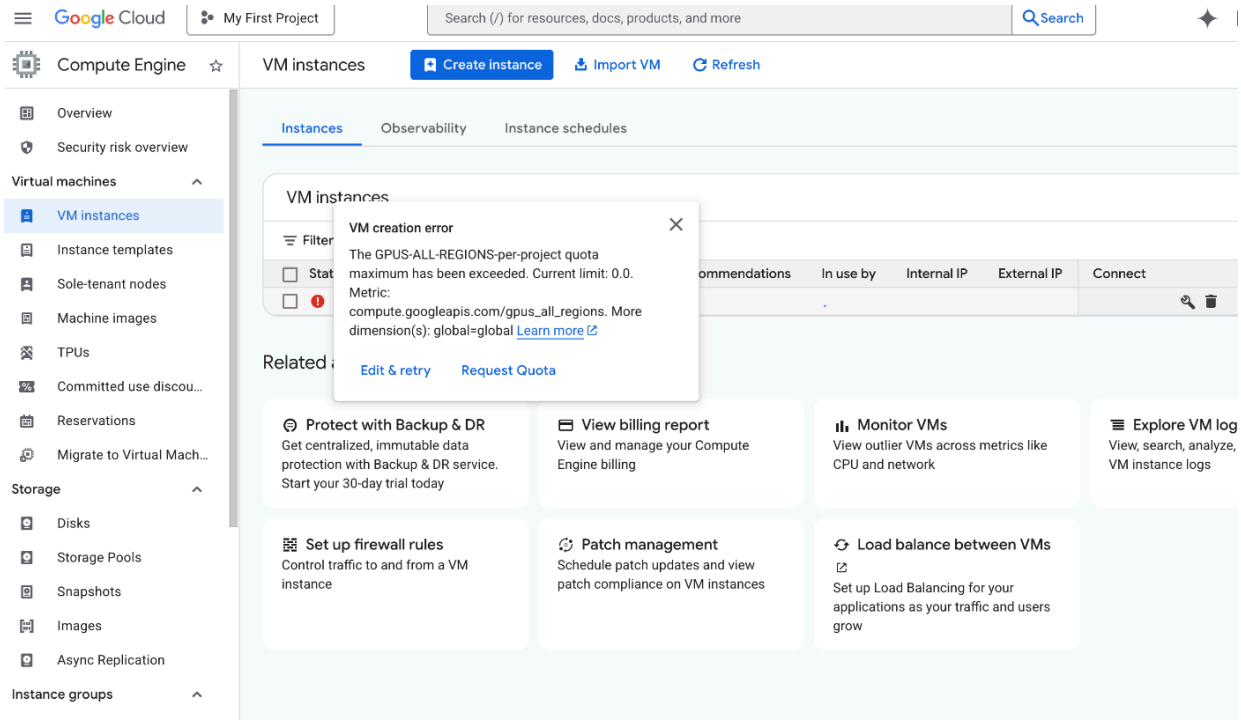

Figure 3. No Cloud GPU Resources on Google Compute Engine

Part 4: Optimizing CUDA Code

In this part, we use tiled matrix multiplication with shared memory to reduce global memory accesses. The full code is attached:

```
// matrix_tiled_cuda.cu
// Tiled CUDA matmul using shared memory (TILE_WIDTH=16) + kernel timing (ms)
// Compile: nvcc -O2 matrix_tiled_cuda.cu -o matrix_tiled_cuda.exe
// Run:     matrix_tiled_cuda.exe 1024

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

#define TILE_WIDTH 16

__global__ void matrixMultiplyTiled(float *A, float *B, float *C, int N) {
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0.0f;

    int numTiles = (N + TILE_WIDTH - 1) / TILE_WIDTH;
    for (int m = 0; m < numTiles; ++m) {
        // Load A tile
        int aCol = m * TILE_WIDTH + tx;
        if (Row < N && aCol < N)
            ds_A[ty][tx] = A[Row * N + aCol];
        else
            ds_A[ty][tx] = 0.0f;

        // Load B tile
        int bRow = m * TILE_WIDTH + ty;
        if (Col < N && bRow < N)
            ds_B[ty][tx] = B[bRow * N + Col];
        else
            ds_B[ty][tx] = 0.0f;

        __syncthreads();

        // Compute partial dot product for this tile
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += ds_A[ty][k] * ds_B[k][tx];

        __syncthreads();
    }

    if (Row < N && Col < N)
        C[Row * N + Col] = Pvalue;
}
```

```
int main(int argc, char** argv) {
    int N = 1024;
    if (argc > 1) N = atoi(argv[1]);

    size_t bytes = (size_t)N * (size_t)N * sizeof(float);

    // Host memory
    float *hA = (float*)malloc(bytes);
    float *hB = (float*)malloc(bytes);
    float *hC = (float*)malloc(bytes);
    if (!hA || !hB || !hC) {
        printf("Host malloc failed\n");
        return 1;
    }

    // Initialize A and B (simple)
    for (int i = 0; i < N * N; i++) {
        hA[i] = 1.0f;
        hB[i] = 1.0f;
    }

    // Device memory
    float *dA = NULL, *dB = NULL, *dC = NULL;
    if (cudaMalloc((void**)&dA, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dB, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dC, bytes) != cudaSuccess) {
        printf("cudaMalloc failed\n");
        return 1;
    }

    // Copy to device
    if (cudaMemcpy(dA, hA, bytes, cudaMemcpyHostToDevice) != cudaSuccess ||
        cudaMemcpy(dB, hB, bytes, cudaMemcpyHostToDevice) != cudaSuccess) {
        printf("cudaMemcpy H2D failed\n");
        return 1;
    }

    // Launch config (matches TILE_WIDTH)
    dim3 block(TILE_WIDTH, TILE_WIDTH);
    dim3 grid((N + TILE_WIDTH - 1) / TILE_WIDTH, (N + TILE_WIDTH - 1) / TILE_WIDTH);

    // Timing with CUDA events (kernel time only)
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Warmup
    matrixMultiplyTiled<<<grid, block>>>(dA, dB, dC, N);
    cudaDeviceSynchronize();

    cudaEventRecord(start);
    matrixMultiplyTiled<<<grid, block>>>(dA, dB, dC, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
```

```
    float ms = 0.0f;
    cudaEventElapsedTime(&ms, start, stop);

    // Minimal error check
    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("Kernel launch error: %s\n", cudaGetErrorString(err));
        return 1;
    }

    // Copy result back
    if (cudaMemcpy(hC, dC, bytes, cudaMemcpyDeviceToHost) != cudaSuccess) {
        printf("cudaMemcpy D2H failed\n");
        return 1;
    }

    // Output (time + one value)
    printf("N=%d, time_ms=%f, C0=%f\n", N, ms, hC[0]);

    // Cleanup
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    free(hA);
    free(hB);
    free(hC);

    return 0;
}
```

We skip attaching the script file here since they are the same as previous except file name change

Table3. Runtime Summary for Optimized CUDA Based Matrix Multiplication

	Run #\N	256	512	1024	2048	4096
1	Run 1	0.0379	0.218	1.627	14.328	105.831
2	Run 2	0.0387	0.217	1.627	13.26	99.715
3	Run 3	0.0387	0.218	1.627	13.306	106.565
4	Run 4	0.0387	0.217	1.636	12.794	108.037
5	Run 5	0.0387	0.218	1.626	14.035	100.984
6	Average	0.03854	0.2176	1.6286	13.5446	104.2264

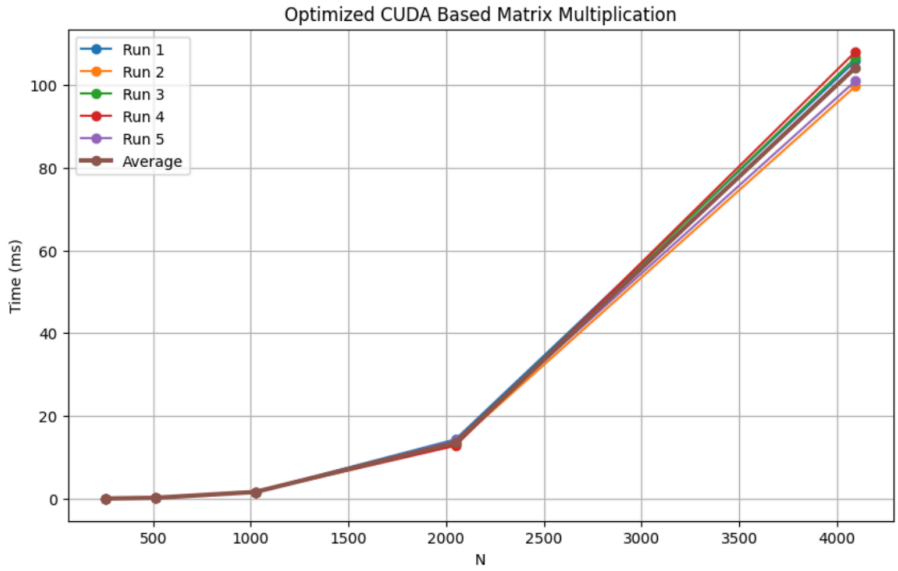

Figure 4. Line Graph for Optimized CUDA Based Matrix Multiplication

Part 5: Performance Comparison

Based on the data collection from previous parts, we can conclude the performance and speedup in following tables:

Table 4. Performance Comparison

	Implementation	N=256	N=512	N=1024	N=2048	N=4096
1	CPU (C)	0.0148 s (14.8 ms)	0.151 s (151 ms)	2.585 s (2584.6 ms)	31.020 s (31020 ms)	655.875 s (655875 ms)
2	Naive CUDA	0.0458 ms	0.2776 ms	2.1998 ms	17.4722 ms	141.1908 ms
3	Optimized CUDA	0.03855ms	0.2176 ms	1.6286 ms	13.5446 ms	104.2264 ms

Table 5: Speedup Summary

	Speedup	N=256	N=512	N=1024	N=2048	N=4096
1	CPU / Naive CUDA	323.14×	543.95×	1174.92×	1775.39×	4645.31×
2	CPU / Optimized CUDA	384.02×	693.93×	1587.01×	2290.21×	6292.79×
3	Naive CUDA / Optimized CUDA (ref)	1.19x	1.28x	1.35x	1.29x	1.35x

Part 6: Using cuBLAS Library

In this part, we use built-in cuBLAS library for matrix multiplication tasks. cuBLAS library is a highly optimized CUDA library that

can outperform normal customized/semi-customized CUDA program/library.

```c
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>

int main(int argc, char** argv) {
    int N = 1024;
    if (argc > 1) N = atoi(argv[1]);

    size_t bytes = (size_t)N * (size_t)N * sizeof(float);

    // Host memory
    float *hA = (float*)malloc(bytes);
    float *hB = (float*)malloc(bytes);
    float *hC = (float*)malloc(bytes);
    if (!hA || !hB || !hC) {
        printf("Host malloc failed\n");
        return 1;
    }

    // Initialize A and B
    for (int i = 0; i < N * N; i++) {
        hA[i] = 1.0f;
        hB[i] = 1.0f;
    }

    // Device memory
    float *dA = NULL, *dB = NULL, *dC = NULL;
    if (cudaMalloc((void**)&dA, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dB, bytes) != cudaSuccess ||
        cudaMalloc((void**)&dC, bytes) != cudaSuccess) {
        printf("cudaMalloc failed\n");
        return 1;
    }

    // Copy to device
    if (cudaMemcpy(dA, hA, bytes, cudaMemcpyHostToDevice) != cudaSuccess ||
        cudaMemcpy(dB, hB, bytes, cudaMemcpyHostToDevice) != cudaSuccess) {
        printf("cudaMemcpy H2D failed\n");
        return 1;
    }

    // cuBLAS handle
    cublasHandle_t handle;
    if (cublasCreate(&handle) != CUBLAS_STATUS_SUCCESS) {
        printf("cublasCreate failed\n");
        return 1;
    }

    // SGEMM params: C = alpha * op(A) * op(B) + beta * C
    // NOTE: cuBLAS assumes column-major storage by default.
    // To treat our row-major arrays as-is, we compute:
```

```
    //   C^T = B^T * A^T
    // which corresponds to swapping A/B in the call below.
    float alpha = 1.0f;
    float beta  = 0.0f;

    // Timing (GPU time for the cuBLAS call)
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Warmup
    cublasSgemm(handle,
                CUBLAS_OP_N, CUBLAS_OP_N,
                N, N, N,
                &alpha,
                dB, N,   // B first
                dA, N,   // A second
                &beta,
                dC, N);
    cudaDeviceSynchronize();

    cudaEventRecord(start);
    cublasSgemm(handle,
                CUBLAS_OP_N, CUBLAS_OP_N,
                N, N, N,
                &alpha,
                dB, N,
                dA, N,
                &beta,
                dC, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float ms = 0.0f;
    cudaEventElapsedTime(&ms, start, stop);

    // Copy back
    if (cudaMemcpy(hC, dC, bytes, cudaMemcpyDeviceToHost) != cudaSuccess) {
        printf("cudaMemcpy D2H failed\n");
        return 1;
    }

    // Print time + one value
    printf("N=%d, time_ms=%f, C0=%f\n", N, ms, hC[0]);

    // Cleanup
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    cublasDestroy(handle);
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
    free(hA);
    free(hB);
    free(hC);
```

```
    return 0;
}
```

Table 6. Runtime Summary for cuBLAS Based Matrix Multiplication

	Run #\N	256	512	1024	2048	4096
1	Run 1	0.0256	0.0676	0.251	1.678	13.437
2	Run 2	0.0253	0.0614	0.25	1.677	14.242
3	Run 3	0.0399	0.0614	0.25	1.676	13.056
4	Run 4	0.0387	0.0625	0.251	1.68	13.006
5	Run 5	0.0286	0.0614	0.25	1.695	13.023
6	Average	0.03162	0.06286	0.2504	1.6812	13.3528

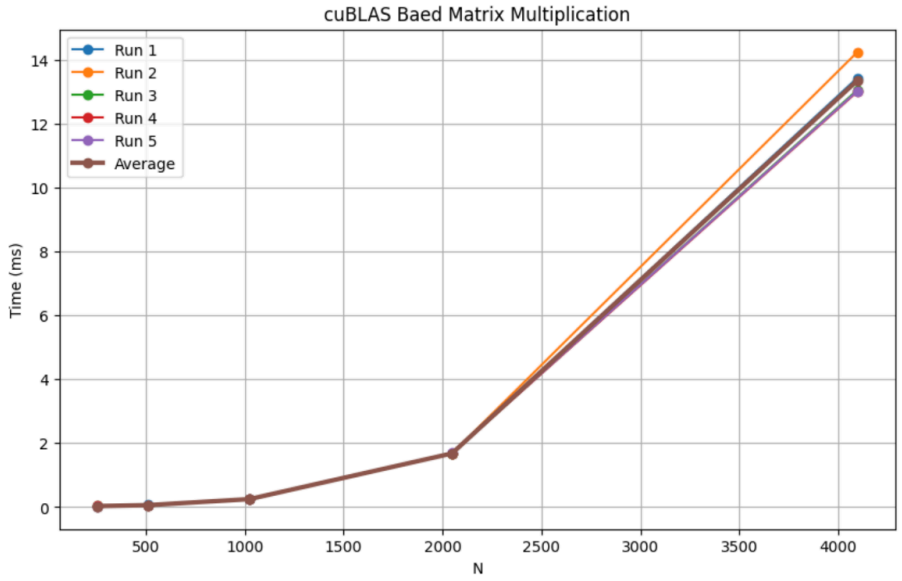

Figure 5. Line Graph for cuBLAS Based Matrix Multiplication

Part 7(a): Analysis Questions

HOW DOES PERFORMANCE CHANGE AS MATRIX SIZE INCREASES?

As matrix size increases, the runtime rises very quickly because the matrix multiplication without optimization does O(N^3). This trend is more severe in CPU based calculation and the runtime increases exponentially. As for the CUDA accelerated performance, the curves look closer to ideal cubix scaling at larger sizes. cuBLAS scales best overall because of high optimization for this library.

AT WHAT POINT DOES THE GPU SIGNIFICANTLY OUTPERFORM THE CPU?

In our measurement, the GPU significantly outperform the CPU at every tested size starting with 256. Normally, when the size is small, due to `cudamemcpy` or `cudamalloc` (CUDA unique processes), it can create higher overhead compared to CPU instant on-chip computation. However, this phenomenon is not shown during our tests.

HOW MUCH SPEEDUP IS GAINED BY TILING OPTIMIZATION VS. NAÏVE CUDA?

Tiling (shared-memory) optimization provides a consistent speedup over naïve CUDA by reducing expensive global memory accesses and reusing tiles of A and B inside a block. Quantitatively, Naive/Optimized speedup ranges from about 1.19× (N=256) to about 1.35× (N=1024 and N=4096), with values around 1.28–1.29× at N=512 and N=2048. In other words, tiling makes the kernel roughly 19%–35% faster, and the benefit is most visible once matrices are large enough that global memory traffic becomes a dominant cost.

HOW CLOSE IS YOUR OPTIMIZED KERNEL TO CUBLAS PERFORMANCE?

The optimized kernel is still notably slower than cuBLAS, especially as N grows. Comparing runtimes, the optimized kernel is only about 1.22× slower than cuBLAS at N=256, but the gap widens to roughly 3.46× at N=512 and about 6–8× slower for N=1024 through N=4096 (e.g., ~6.5× at N=1024, ~8.1× at N=2048, ~7.8× at N=4096). This indicates the tiling optimization captures the basic idea of reuse, but cuBLAS is operating much closer to hardware limits at large sizes.

WHY MIGHT CUBLAS STILL OUTPERFORM HAND-WRITTEN KERNELS?

cuBLAS can still outperform a hand-written kernel because it's a professionally engineered library that's been tuned and refined across many GPU generations. It typically makes better use of the hardware by carefully organizing computation and memory access to reduce wasted work, minimize slow memory traffic etc. It also benefits from extensive low-level tuning and internal heuristics that pick efficient strategies for different matrix sizes and hardware configurations. In contrast, a custom kernel written for a lab is usually designed for clarity and correctness first, so it rarely matches the level of optimization, adaptability, and hardware-specific refinement found in cuBLAS.

Part 7(b): Creating a Shared Library and Using it in Python

We use the existing code and command for compiling in lab manual but make some changes according to Windows. environment. We finish the python call CUDA library as indicated in following figure.

```
PS D:\ee533\cuda\part7> nvcc -O2 -shared -Xcompiler "/MD" libmatrix.cu -o libmatrix.so
libmatrix.cu
tmpxft_00004214_00000000-10_libmatrix.cudafe1.cpp
   Creating library libmatrix.lib and object libmatrix.exp
PS D:\ee533\cuda\part7> python3 .\cuda_lib_call.py
Python call to CUDA library completed in 0.1695 seconds
PS D:\ee533\cuda\part7>
```

Figure 6. Python Call CUDA Library

We further implement custom function (i.e. convolution function) to the shared library with both API call for CPU and GPU for rough performance comparison:

```
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#ifdef _WIN32
  #define EXPORT extern "C" __declspec(dllexport)
#else
  #define EXPORT extern "C"
#endif
```

```
// Simple CUDA error check (minimal)
static void checkCuda(cudaError_t e, const char* msg) {
    if (e != cudaSuccess) {
        printf("CUDA error (%s): %s\n", msg, cudaGetErrorString(e));
    }
}


__global__ void conv2d_u8_same_kernel(const unsigned char* img,
                                      const float* ker,
                                      float* out,
                                      int M, int N)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x; // col
    int y = blockIdx.y * blockDim.y + threadIdx.y; // row
    if (x >= M || y >= M) return;

    int r = N / 2;
    float sum = 0.0f;

    for (int ky = 0; ky < N; ky++) {
        for (int kx = 0; kx < N; kx++) {
            int iy = y + ky - r;
            int ix = x + kx - r;
            if (iy >= 0 && iy < M && ix >= 0 && ix < M) {
                unsigned char pix = img[iy * M + ix];
                sum += ((float)pix) * ker[ky * N + kx];
            }
        }
    }
    out[y * M + x] = sum;
}

//Non-CUDA reference implementation: CPU convolution
EXPORT float cpu_convolve_u8(const unsigned char* image, const float* kernel,
                             float* out, int M, int N) {
#ifdef _WIN32
    LARGE_INTEGER freq, t0, t1;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&t0);
#endif

    int r = N / 2;
    for (int y = 0; y < M; y++) {
        for (int x = 0; x < M; x++) {
            float sum = 0.0f;
            for (int ky = 0; ky < N; ky++) {
                for (int kx = 0; kx < N; kx++) {
                    int iy = y + ky - r;
                    int ix = x + kx - r;
                    if (iy >= 0 && iy < M && ix >= 0 && ix < M) {
                        unsigned char pix = image[iy * M + ix];
                        sum += ((float)pix) * kernel[ky * N + kx];
                    }
                }
```

```
            }
            out[y * M + x] = sum;
        }
    }
#ifdef _WIN32
    QueryPerformanceCounter(&t1);
    double ms = (double)(t1.QuadPart - t0.QuadPart) * 1000.0 / (double)freq.QuadPart;
    return (float)ms;
#else
    return -1.0f;
#endif
}

EXPORT float gpu_convolve_u8(const unsigned char* h_img,
                             const float* h_ker,
                             float* h_out,
                             int M, int N)
{
    if (!h_img || !h_ker || !h_out || M <= 0 || N <= 0) return -1.0f;

    size_t imgBytes = (size_t)M * (size_t)M * sizeof(unsigned char);
    size_t kerBytes = (size_t)N * (size_t)N * sizeof(float);
    size_t outBytes = (size_t)M * (size_t)M * sizeof(float);

    unsigned char* d_img = NULL;
    float* d_ker = NULL;
    float* d_out = NULL;

    cudaError_t e;

    e = cudaMalloc((void**)&d_img, imgBytes); checkCuda(e, "cudaMalloc d_img");
    e = cudaMalloc((void**)&d_ker, kerBytes); checkCuda(e, "cudaMalloc d_ker");
    e = cudaMalloc((void**)&d_out, outBytes); checkCuda(e, "cudaMalloc d_out");

    e = cudaMemcpy(d_img, h_img, imgBytes, cudaMemcpyHostToDevice); checkCuda(e, "H2D
    e = cudaMemcpy(d_ker, h_ker, kerBytes, cudaMemcpyHostToDevice); checkCuda(e, "H2D

    dim3 block(16, 16);
    dim3 grid((M + block.x - 1) / block.x,
              (M + block.y - 1) / block.y);

    // Timing: kernel only
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Optional warm-up (helps stabilize measurements)
    conv2d_u8_same_kernel<<<grid, block>>>(d_img, d_ker, d_out, M, N);
    cudaDeviceSynchronize();

    cudaEventRecord(start);
    conv2d_u8_same_kernel<<<grid, block>>>(d_img, d_ker, d_out, M, N);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
```

```
    float ms = 0.0f;
    cudaEventElapsedTime(&ms, start, stop);

    // Catch kernel launch/runtime errors
    e = cudaGetLastError(); checkCuda(e, "kernel launch");
    e = cudaDeviceSynchronize(); checkCuda(e, "kernel sync");

    e = cudaMemcpy(h_out, d_out, outBytes, cudaMemcpyDeviceToHost); checkCuda(e, "D2H

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    cudaFree(d_img);
    cudaFree(d_ker);
    cudaFree(d_out);

    return ms;
}
```

Also the python code to call the custom library:

```
import ctypes, numpy as np, time

lib = ctypes.cdll.LoadLibrary("./libcustom.so")

lib.gpu_convolve_u8.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.uint8,   ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    ctypes.c_int, ctypes.c_int
]
lib.gpu_convolve_u8.restype = ctypes.c_float  # returns kernel ms

lib.cpu_convolve_u8.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.uint8,   ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    np.ctypeslib.ndpointer(dtype=np.float32, ndim=1, flags="C_CONTIGUOUS"),
    ctypes.c_int, ctypes.c_int
]
lib.cpu_convolve_u8.restype = ctypes.c_float  # returns kernel ms

M = 2**12
N = 5
img = (np.random.rand(M, M) * 255).astype(np.uint8)

# example edge filter (Sobel X)
ker = np.array([[-1,0,1],[-2,0,2],[-1,0,1]], dtype=np.float32)

out = np.zeros((M, M), dtype=np.float32)

t0 = time.time()
ms_gpu = lib.gpu_convolve_u8(img.ravel(), ker.ravel(), out.ravel(), M, N)
t1 = time.time()
```

```
t2 = time.time()
# verify correctness with simple CPU convolution
ms_cpu = lib.cpu_convolve_u8(img.ravel(), ker.ravel(), out.ravel(), M, N)
t3 = time.time()

print(f"gpu_ms={ms_gpu:.3f}, python_total_s={t1-t0:.4f}")
print(f"cpu_ms={ms_cpu:.3f}, python_total_s={t3-t2:.4f}")
```

Here we do not use images as the inputs but use random value to demonstrate the computation. We run 5 times of the same test python program:

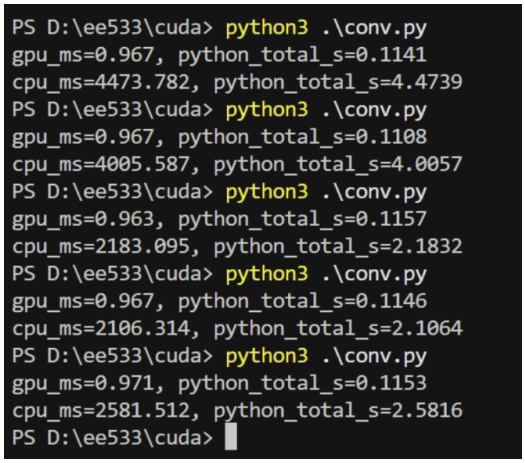

Figure 7. 5 Runs of conv.py with CPU and GPU Runtimes