Laboratory 3: CUDA Program and Custom Python Library
Instructor: Young H. Cho, Ph.D.

Objectives

By the end of this lab, you will be able to:

1. Write and execute a C program performing large matrix operations on a CPU.
2. Measure CPU execution performance with different matrix sizes.
3. Port the program to CUDA and execute it on a GPU.
4. Deploy a GPU-enabled virtual machine on Google Cloud and run CUDA programs.
5. Optimize CUDA code for better GPU performance.
6. Compare performance results between CPU, naïve CUDA, optimized CUDA, and cuBLAS implementations.
7. Analyze performance scaling as the problem size grows.
8. Create shared libraries using CUDA and make use of the acceleration from Python.

Part 1: Matrix Multiplication on the CPU

Step 1.1: Write a C Program for Matrix Multiplication

You need to implement a program that does a **matrix multiplication** in C

Example Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void matrixMultiplyCPU(float *A, float *B, float *C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}

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

    float *A = (float *)malloc(size);
    float *B = (float *)malloc(size);
    float *C = (float *)malloc(size);

    for (int i = 0; i < N * N; i++) {
        A[i] = rand() % 100 / 100.0f;
        B[i] = rand() % 100 / 100.0f;
```

```
    }

    clock_t start = clock();
    matrixMultiplyCPU(A, B, C, N);
    clock_t end = clock();

    double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
    printf("CPU execution time (N=%d): %f seconds\n", N, elapsed);

    free(A); free(B); free(C);
    return 0;
}
```

Step 1.2: Compile and Run

```
gcc matrix_cpu.c -o matrix_cpu -O2
./matrix_cpu 512
./matrix_cpu 1024
./matrix_cpu 2048
```

More measurements should be made to record and graph runtimes for several **N** values. You may want to write a script to automate this data collection.

Include these in your laboratory report.

Part 2: Introduction to CUDA Programming

Step 2.1: Naïve CUDA Kernel

Implement a CUDA kernel where each thread computes one element of the output matrix.

```
__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;
    }
}
```

Run the CUDA program with multiple matrix sizes (512, 1024, 2048, etc.) and measure execution time as you did in Part 1. If you have an Nvidia GPU that supports CUDA on your computer.

[Optional] You may set up the development kit on your computer to execute and make these measurements. On an Intel-based computer with an Nvidia GPU, you can install CUDA 13 on Windows or Linux by following the instructions on

https://developer.nvidia.com/cuda-downloads
While this step is optional, you may find that that the additional hardware and the skills you gain going through this will be very useful and convenient for your career in the long run.

Part 3: Running CUDA on Google Cloud

Step 3.1: Provision a GPU Instance

- Create a VM in Google Cloud Compute Engine with:
  - **GPU**: NVIDIA Tesla T4 or V100
  - **Machine type**: n1-standard-8 or higher
  - **OS**: Ubuntu 20.04 LTS or other supported Linux

Step 3.2: Install CUDA Toolkit

```
sudo apt-get update
sudo apt-get install -y nvidia-driver-470
sudo apt-get install -y nvidia-cuda-toolkit
nvcc --version
```

Step 3.3: Run Naïve CUDA Program

```
nvcc matrix_gpu.cu -o matrix_gpu
./matrix_gpu 1024
```

Record runtimes for different N values.

Part 4: Optimizing CUDA Code

Step 4.1: Shared Memory Tiling

Introduce tiling with shared memory to reduce global memory accesses.

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

```
        if (Col < N && (m*TILE_WIDTH+ty) < N)
            ds_B[ty][tx] = B[(m*TILE_WIDTH + ty) * N + Col];
        else
            ds_B[ty][tx] = 0.0f;

        __syncthreads();

        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;
}
```

Measure runtimes with **N = 512, 1024, 2048**.

Part 5: Performance Comparison

Create a table:

Implementation	**N=512**	**N=1024**	**N=2048**	**…**
CPU (C)	X sec	Y sec	Z sec	…
Naïve CUDA	A ms	B ms	C ms	…
Optimized CUDA	D ms	E ms	F ms	…

Compute **speedup = CPU time / GPU time**.

As you increase the size of your data, you should notice the overhead in using a GPU/DPU interface card connected as one of the peripherals of the CPU. Consider the impact of the overhead to define the constraints of the functions and optimize the overall performance of the program.

Part 6: Using cuBLAS Library

Use **cuBLAS** (already included with CUDA Toolkit) to perform matrix multiplication.

```
cublasSgemm(handle,
            CUBLAS_OP_N, CUBLAS_OP_N,
            N, N, N,
            &alpha,
            d_B, N,
            d_A, N,
            &beta,
            d_C, N);
```

Measure runtimes for **N = 512, 1024, 2048, etc.** and add them to the table:

Implementation	N=512	N=1024	N=2048	…
CPU (C)	X sec	Y sec	Z sec	…
Naïve CUDA	A ms	B ms	C ms	…
Optimized CUDA	D ms	E ms	F ms	…
cuBLAS	G ms	H ms	I ms	…

Part 7: Analysis Questions

1. How does performance change as matrix size increases?
2. At what point does the GPU significantly outperform the CPU?
3. How much speedup is gained by tiling optimization vs. naïve CUDA?
4. How close is your optimized kernel to cuBLAS performance?
5. Why might cuBLAS still outperform hand-written kernels?

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

In this part, compile your CUDA matrix multiplication kernel from Part 4 into a shared library (.so file) and call it from Python using ctypes.

Step 7.1: Write a Library Interface

Modify the CUDA code to expose a function suitable for Python to call:

```
#include <cuda_runtime.h>
#include <stdio.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 = 0.0;
    for (int m = 0; m < (N + TILE_WIDTH - 1) / TILE_WIDTH; ++m) {
        if (Row < N && (m*TILE_WIDTH+tx) < N)
            ds_A[ty][tx] = A[Row * N + m * TILE_WIDTH + tx];
        else
            ds_A[ty][tx] = 0.0f;

        if (Col < N && (m*TILE_WIDTH+ty) < N)
            ds_B[ty][tx] = B[(m*TILE_WIDTH + ty) * N + Col];
        else
            ds_B[ty][tx] = 0.0f;
```

```
        __syncthreads();

        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;
}

// Exposed C function for Python
extern "C" void gpu_matrix_multiply(float *h_A, float *h_B, float *h_C, int
N) {
    size_t size = N * N * sizeof(float);
    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);

    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
    dim3 dimGrid((N + TILE_WIDTH - 1) / TILE_WIDTH, (N + TILE_WIDTH - 1) /
TILE_WIDTH);

    matrixMultiplyTiled<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Step 7.2: Compile as a Shared Library

```
nvcc -Xcompiler -fPIC -shared matrix_lib.cu -o libmatrix.so
```

This creates a dynamic library `libmatrix.so` that Python can call.

Step 7.3: Python Code Using the Library

Write a Python script to call the CUDA library using `ctypes`:

```
import ctypes
import numpy as np
import time

# Load shared library
lib = ctypes.cdll.LoadLibrary("./libmatrix.so")

# Define argument types
```

```
lib.gpu_matrix_multiply.argtypes = [
    np.ctypeslib.ndpointer(dtype=np.float32, 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
]

N = 1024
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)
C = np.zeros((N, N), dtype=np.float32)

start = time.time()
lib.gpu_matrix_multiply(A.ravel(), B.ravel(), C.ravel(), N)
end = time.time()

print(f"Python call to CUDA library completed in {end - start:.4f} seconds")
```

Part 2: Adding Custom Functions to the Shared Library

Step 1: Convolution Function

Write a C code for an N by N (smaller filter matrix) to an M by M (larger image matrix) convolution function. Assume that each pixel of the matrix is a single unsigned integer, making the matrices monochrome images. You may create, generate, and convert images using various online and offline image tools, as well as MATLAB and SciPy (saving as 32-bit PNG or TIFF may result in these matrices). Filters may be designed for different types of image processing, including edge detection. For more information, you may search the web. (i.e., https://en.wikipedia.org/wiki/Kernel_(image_processing))

Create a few image processing filters, including edge detection. Collect a sample of images. Then test your convolution function on the images using the filters and demonstrate that your function works properly with images of the results included in your report.

Collect performance numbers for at least images and templates of 3 different Ns and 3 different Ms.

Step 2: Porting the Convolution Function to CUDA

Port your C code to CUDA. Collect the performance numbers for the same images and templates. Compare the performances.

Step 2: Adding and Using the Convolution to your Custom Python Library

Add the CUDA-accelerated functions of your choice to the shared library. Then write Python code(s) that use the newly created library functions to accelerate a program. Compare the performance of your CUDA-accelerated Python program against a non-accelerated C program and a directly executed CUDA executable that performs the same function.

Deliverables

1. **Source code**: CPU, naïve CUDA, optimized CUDA, cuBLAS, Shared Libraries, and Python codes.
2. **Graphs**: plot execution time vs. matrix size for All CPU and GPU implementations.
3. **Report**
    - Observations on performance scaling.
    - Analysis of overhead associated with using GPU/DPU accelerator
    - Reflections on optimization trade-offs.
    - List and description of your shared library for Python