Homework # 4

Rohan Kapur, ECPE 293A Spring 2025

DUE: March 12, 2025

1. Naive matrix multiplication implementation on CPU vs GPU, with a CUDA block size of 25x25=625<1024 threads, which ensures that there aren't any idle threads for processing matrix entries in either block dimension when the square matrix dimension is a power of 10.

Matrix Size	CPU time (ms)	Naive GPU time (ms)		
		H2D transfer	Kernel	D2H transfer
1000x1000	31.29	15.31	0.53	56.99
2000x2000	114.90	45.96	0.50	409.86
4000x4000	790.10	173.53	2.02	2555.74
8000x8000	6058.38	692.98	2.14	19766.34
16000x16000	47313.05	2778.62	1.23	205529.16

By *far* the biggest bottleneck is the time for transferring data to and from the GPU. Otherwise, the actual matrix multiplication operation performs orders of magnitude faster on the GPU than on the CPU.

```
import time
import numpy as np
import pycuda.autoinit
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import argparse

naive_kernel = SourceModule(
    r"""
    #include <stdio.h>
    __global__ void naive_matrix_mult(double *d_A, double *d_B, double *d_C, int
        width) {
        double temp = 0;
        int row = threadIdx.x + blockIdx.x*blockDim.x;
        int col = threadIdx.y + blockIdx.y*blockDim.y;
        if (row < width && col < width) {
            temp = 0;
            for (int i = 0; i < width; i++) {
                temp += d_A[row*width + i] * d_B[i*width + col];
            }
            d_C[row*width + col] = temp;
        }
    }
    """
```

```
)


def matrix_mult_gpu(A: np.ndarray, B: np.ndarray):
    """Perform naive matrix multiplication in parallel on the selected GPU device

    Args:
        A (np.ndarray): First square matrix to multiply
        B (np.ndarray): Second square matrix to multiply

    Returns:
        np.ndarray: Result of matrix multiplication
    """

    # Assertion checks
    assert A.shape[0] == A.shape[1], "Matrix A is not square!"
    assert B.shape[0] == B.shape[1], "Matrix B is not square!"
    assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"

    matrix_size = A.shape[0]

    # Send input array data to GPU device and time it
    start_htod = time.time()
    A_flatten = A.flatten()
    A_gpu = cuda.mem_alloc(A_flatten.nbytes)
    cuda.memcpy_htod(A_gpu, A_flatten)
    B_flatten = B.flatten()
    B_gpu = cuda.mem_alloc(B_flatten.nbytes)
    cuda.memcpy_htod(B_gpu, B_flatten)
    time_htod = (time.time() - start_htod) * 1000
    print(f"GPU host-to-device transfer time: {time_htod:.2f}ms")

    # Allocate space on device for output
    C = np.empty_like(B).flatten()
    C_gpu = cuda.mem_alloc(C.nbytes)

    # Get device kernel and execute it
    func = naive_kernel.get_function("naive_matrix_mult")
    # Create 2D blocks of 32x32 threads (max 1024 threads allowed in block)
    # Create grid of ceil(matrix_size // block_width) x ceil(matrix_size //
        block_height) blocks
    block_width = 25
    block_height = 25
    assert block_width * block_height <= 1024, "Max 1024 threads allowed in block!"
    # Proper way to compute CEIL for grid dimensions
    grid_width = (matrix_siz + block_width - 1) // block_width
    grid_height = (matrix_size + block_height - 1) // block_height
    start_kernel = time.time()
    func(
        A_gpu,
        B_gpu,
        C_gpu,
        np.int32(matrix_size),
        block=(block_width, block_height, 1),
```

```
        grid=(grid_width, grid_height),
    )
    time_kernel = (time.time() - start_kernel) * 1000
    print(f"GPU kernel execution time: {time_kernel:.2f}ms")

    # Retrieve output data from the GPU and time it
    start_dtoh = time.time()
    cuda.memcpy_dtoh(C, C_gpu)
    time_dtoh = (time.time() - start_dtoh) * 1000
    print(f"GPU device-to-host transfer time: {time_dtoh:.2f}ms")

    total_time = time_htod + time_kernel + time_dtoh
    print(f"Total GPU time: {total_time:.2f}ms")

    # Return matrix multiplication result as properly shaped matrix
    return C.reshape(matrix_size, matrix_size)


def matrix_mult_cpu(A: np.ndarray, B: np.ndarray):
    """Perform Numpy matrix multiplication given the two np.ndarray matrices

    Args:
        A (np.ndarray): The first square matrix to multiply
        B (np.ndarray): The second square matrix to multiply

    Returns:
        np.ndarray: Result of matrix multiplication
    """

    # Assertion checks
    assert A.shape[0] == A.shape[1], "Matrix A is not square!"
    assert B.shape[0] == B.shape[1], "Matrix B is not square!"
    assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"

    # Perform multiplication, time it, & return result
    time_start = time.time()
    C = A @ B
    cpu_time = 1000 * (time.time() - time_start)
    print(f"CPU multiplication time: {cpu_time:.2f}ms")
    return C


def main():
    # Initialize argument parser
    parser = argparse.ArgumentParser(
        description="Compare naive matrix multi[lication between GPU and CPU"
    )
    parser.add_argument(
        "--matrix_size",
        type=int,
        required=True,
        help="Size of the square matrices being multiplied",
    )
```

```python
    # Get required command line argument
    args = parser.parse_args()
    matrix_size = args.matrix_size

    # Initialize CUDA driver
    cuda.init()

    # Get number of GPU's
    num_gpus = cuda.Device.count()
    print(f"Number of available GPUs: {num_gpus}")

    # Prompt for Pacific ID
    user_id = int(input("Please enter your Pacific ID: "))

    # Determine which GPU to use
    gpu_index = user_id % num_gpus
    print(f"Using GPU index: {gpu_index}")
    selected_gpu = cuda.Device(gpu_index)
    print(f"Selected GPU: {selected_gpu.name()}")

    # Create numpy square matrices of requested size
    A = np.random.randn(matrix_size, matrix_size).astype(np.double)
    B = np.random.randn(matrix_size, matrix_size).astype(np.double)

    # Perform matrix multiplication on the CPU
    cpu_result = matrix_mult_cpu(A, B)

    # Perform GPU accelerated matrix multiplication
    gpu_result = matrix_mult_gpu(A, B)

    # Verify results are the same
    if np.allclose(cpu_result, gpu_result):
        print("Results are the same!")
    else:
        print("Results are different!")
        print(f"CPU result: {cpu_result}")
        print(f"GPU result: {gpu_result}")


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

2. Optimized shared memory implementation of matrix multiplication on the GPU using same CUDA setup as the above

Matrix Size	CPU time (ms)	Optimized GPU time (ms)		
		H2D transfer	Kernel	D2H transfer
1000x1000	34.37	35.16	0.55	33.75
2000x2000	119.88	47.1	0.56	219.33
4000x4000	791.36	173.89	2.28	1347.64
8000x8000	6118.04	708.13	2.22	9830.55
16000x16000	47355.23	2825.87	1.32	78037.28

The most significant difference here from the previous naive implementation of matrix product is the significant reduction in the Device-to-Host data transfer time across the board, which most likely has to do with the shared memory being used, which lies on-board and closer to the thread blocks/CUDA cores. This takes fewer clock cycles to access and therefore would take less time to fetch data from for returning to the host. Otherwise, the kernel execution time remained about the same as with the naive implementation.

```python
import time
import numpy as np
import pycuda.autoinit
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import argparse

opt_kernel = SourceModule(
    r"""
    #define TILEWIDTH 25 // MUST be the same as square thread block width/height!
    __global__ void opt_matrix_mult(double *d_A, double *d_B, double *d_C, int
        width) {
        // Declare shared subtiles of matrix being worked on and shared by all
            threads in the block in shared memory
        __shared__ double Ashared[TILEWIDTH][TILEWIDTH];
        __shared__ double Bshared[TILEWIDTH][TILEWIDTH];
        int bx=blockIdx.x, by=blockIdx.y;
        int tx=threadIdx.x, ty=threadIdx.y;
        int row=tx+bx*TILEWIDTH, col=ty+by*TILEWIDTH;
        double temp = 0; // Store element sums
        // Loop over tiles Ashared and Bshared to compute matrix product elements in
            d_C
        for(int i = 0; i < width/TILEWIDTH; i++) {
            // Collaboratively load correct elements into Ashared and Bshared to
                compute d_C
            Ashared[tx][ty] = d_A[i*TILEWIDTH + row*width + ty];
            Bshared[tx][ty] = d_B[(i*TILEWIDTH+tx)*width + col];
            __syncthreads(); // Ensure all block threads catch up
            // Loop over tiles to compute product entries
            for (int k = 0; k < TILEWIDTH; k++) {
                temp += Ashared[tx][k] * Bshared[k][ty];
            }
            __syncthreads(); // Let all block threads catch up
        }
        d_C[row*width + col] = temp; // Insert element into d_C
    }
    """
)


def opt_matrix_mult_gpu(A: np.ndarray, B: np.ndarray):
    """Perform optimized matrix multiplication in parallel on the selected GPU
        device

    Args:
        A (np.ndarray): First square matrix to multiply
        B (np.ndarray): Second square matrix to multiply
```

```python

    Returns:
        np.ndarray: Result of matrix multiplication
    """

    # Assertion checks
    assert A.shape[0] == A.shape[1], "Matrix A is not square!"
    assert B.shape[0] == B.shape[1], "Matrix B is not square!"
    assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"

    matrix_size = A.shape[0]

    # Send input array data to GPU device and time it
    start_htod = time.time()
    A_flatten = A.flatten()
    A_gpu = cuda.mem_alloc(A_flatten.nbytes)
    cuda.memcpy_htod(A_gpu, A_flatten)
    B_flatten = B.flatten()
    B_gpu = cuda.mem_alloc(B_flatten.nbytes)
    cuda.memcpy_htod(B_gpu, B_flatten)
    time_htod = (time.time() - start_htod) * 1000
    print(f"GPU host-to-device transfer time: {time_htod:.2f}ms")

    # Allocate space on device for output
    C = np.empty_like(B).flatten()
    C_gpu = cuda.mem_alloc(C.nbytes)

    # Get device kernel and execute it
    func = opt_kernel.get_function("opt_matrix_mult")
    # Create 2D blocks of 32x32 threads (max 1024 threads allowed in block)
    # Create grid of ceil(matrix_size // block_width) x ceil(matrix_size // block_height) blocks
    block_width = 25
    block_height = 25
    assert block_width * block_height <= 1024, "Max 1024 threads allowed in block!"
    # Proper way to compute CEIL for grid dimensions
    grid_width = (matrix_size + block_width - 1) // block_width
    grid_height = (matrix_size + block_height - 1) // block_height
    start_kernel = time.time()
    func(
        A_gpu,
        B_gpu,
        C_gpu,
        np.int32(matrix_size),
        block=(block_width, block_height, 1),
        grid=(grid_width, grid_height),
        shared=np.double().nbytes * block_width * block_height,
    )
    time_kernel = (time.time() - start_kernel) * 1000
    print(f"GPU kernel execution time: {time_kernel:.2f}ms")

    # Retrieve output data from the GPU and time it
    start_dtoh = time.time()
    cuda.memcpy_dtoh(C, C_gpu)
```

```python
    time_dtoh = (time.time() - start_dtoh) * 1000
    print(f"GPU device-to-host transfer time: {time_dtoh:.2f}ms")

    # Output total GPU time
    total_time = time_htod + time_kernel + time_dtoh
    print(f"Total GPU time: {total_time:.2f}ms")

    # Return matrix multiplication result as properly shaped matrix
    return C.reshape(matrix_size, matrix_size)


def matrix_mult_cpu(A: np.ndarray, B: np.ndarray):
    """Perform Numpy matrix multiplication given the two np.ndarray matrices

    Args:
        A (np.ndarray): The first square matrix to multiply
        B (np.ndarray): The second square matrix to multiply

    Returns:
        np.ndarray: Result of matrix multiplication
    """

    # Assertion checks
    assert A.shape[0] == A.shape[1], "Matrix A is not square!"
    assert B.shape[0] == B.shape[1], "Matrix B is not square!"
    assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"

    # Perform multiplication, time it, & return result
    time_start = time.time()
    C = A @ B
    cpu_time = 1000 * (time.time() - time_start)
    print(f"CPU multiplication time: {cpu_time:.2f}ms")
    return C


def main():
    # Initialize argument parser
    parser = argparse.ArgumentParser(
        description="Compare naive matrix multi[lication between GPU and CPU"
    )
    parser.add_argument(
        "--matrix_size",
        type=int,
        required=True,
        help="Size of the square matrices being multiplied",
    )

    # Get required command line argument
    args = parser.parse_args()
    matrix_size = args.matrix_size

    # Initialize CUDA driver
    cuda.init()

```

```python
    # Get number of GPU's
    num_gpus = cuda.Device.count()
    print(f"Number of available GPUs: {num_gpus}")

    # Prompt for Pacific ID
    user_id = int(input("Please enter your Pacific ID: "))

    # Determine which GPU to use
    gpu_index = user_id % num_gpus
    print(f"Using GPU index: {gpu_index}")
    selected_gpu = cuda.Device(gpu_index)
    print(f"Selected GPU: {selected_gpu.name()}")

    # Create random numpy square matrices of requested size
    # Sampled from normal distribution with mean of 3 and std deviation of 0.5
    A = 3 + 0.5 * np.random.randn(matrix_size, matrix_size).astype(np.double)
    B = 3 + 0.5 * np.random.randn(matrix_size, matrix_size).astype(np.double)

    # Perform matrix multiplication on the CPU
    cpu_result = matrix_mult_cpu(A, B)

    # Perform GPU accelerated matrix multiplication
    gpu_result = opt_matrix_mult_gpu(A, B)

    # Verify results are the same
    if np.allclose(cpu_result, gpu_result):
        print("Results are the same!")
    else:
        print("Results are different!")
        print(f"CPU result: {cpu_result}")
        print(f"GPU result: {gpu_result}")


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

3. *Active Learning Exercise 15:* With 5 global floating point memory accesses and 13 computations for each of those accesses on lines 11 to 23, the CGMA floating point ratio for listing 10 is **13/5**. Since each float is 4 bytes in memory, the performance achieved by the kernel on a GPGPU device with a bandwidth of 200 GB/s is

$$\frac{200 \text{ GB/s}}{(4 \text{ bytes}) * (5 \text{ floats})} * (13 \text{ operations}) = \mathbf{130\ GFLOPS}$$