

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 $25 \times 25 = 625 < 1024$ 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.

```
1 import time
import numpy as np
3 import pycuda.autoinit
import pycuda.driver as cuda
5 from pycuda.compiler import SourceModule
import argparse
7
naive_kernel = SourceModule(
9     r"""
    #include <stdio.h>
11     __global__ void naive_matrix_mult(double *d_A, double *d_B, double *d_C, int
        width) {
        double temp = 0;
13         int row = threadIdx.x + blockIdx.x*blockDim.x;
        int col = threadIdx.y + blockIdx.y*blockDim.y;
15         if (row < width && col < width) {
            temp = 0;
17             for (int i = 0; i < width; i++) {
                temp += d_A[row*width + i] * d_B[i*width + col];
19             }
            d_C[row*width + col] = temp;
21         }
    }
23     """
```

```

25 )
27 def matrix_mult_gpu(A: np.ndarray, B: np.ndarray):
28     """Perform naive matrix multiplication in parallel on the selected GPU device
29
30     Args:
31         A (np.ndarray): First square matrix to multiply
32         B (np.ndarray): Second square matrix to multiply
33
34     Returns:
35         np.ndarray: Result of matrix multiplication
36     """
37
38     # Assertion checks
39     assert A.shape[0] == A.shape[1], "Matrix A is not square!"
40     assert B.shape[0] == B.shape[1], "Matrix B is not square!"
41     assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"
42
43     matrix_size = A.shape[0]
44
45     # Send input array data to GPU device and time it
46     start_htod = time.time()
47     A_flatten = A.flatten()
48     A_gpu = cuda.mem_alloc(A_flatten.nbytes)
49     cuda.memcpy_htod(A_gpu, A_flatten)
50     B_flatten = B.flatten()
51     B_gpu = cuda.mem_alloc(B_flatten.nbytes)
52     cuda.memcpy_htod(B_gpu, B_flatten)
53     time_htod = (time.time() - start_htod) * 1000
54     print(f"GPU host-to-device transfer time: {time_htod:.2f}ms")
55
56     # Allocate space on device for output
57     C = np.empty_like(B).flatten()
58     C_gpu = cuda.mem_alloc(C.nbytes)
59
60     # Get device kernel and execute it
61     func = naive_kernel.get_function("naive_matrix_mult")
62     # Create 2D blocks of 32x32 threads (max 1024 threads allowed in block)
63     # Create grid of ceil(matrix_size // block_width) x ceil(matrix_size //
64         block_height) blocks
65     block_width = 25
66     block_height = 25
67     assert block_width * block_height <= 1024, "Max 1024 threads allowed in block!"
68     # Proper way to compute CEIL for grid dimensions
69     grid_width = (matrix_size + block_width - 1) // block_width
70     grid_height = (matrix_size + block_height - 1) // block_height
71     start_kernel = time.time()
72     func(
73         A_gpu,
74         B_gpu,
75         C_gpu,
76         np.int32(matrix_size),
77         block=(block_width, block_height, 1),

```

```

77     grid=(grid_width, grid_height),
78     )
79     time_kernel = (time.time() - start_kernel) * 1000
80     print(f"GPU kernel execution time: {time_kernel:.2f}ms")
81
82     # Retrieve output data from the GPU and time it
83     start_dtoh = time.time()
84     cuda.memcpy_dtoh(C, C_gpu)
85     time_dtoh = (time.time() - start_dtoh) * 1000
86     print(f"GPU device-to-host transfer time: {time_dtoh:.2f}ms")
87
88     total_time = time_htod + time_kernel + time_dtoh
89     print(f"Total GPU time: {total_time:.2f}ms")
90
91     # Return matrix multiplication result as properly shaped matrix
92     return C.reshape(matrix_size, matrix_size)
93
94
95 def matrix_mult_cpu(A: np.ndarray, B: np.ndarray):
96     """Perform Numpy matrix multiplication given the two np.ndarray matrices
97
98     Args:
99         A (np.ndarray): The first square matrix to multiply
100        B (np.ndarray): The second square matrix to multiply
101
102     Returns:
103         np.ndarray: Result of matrix multiplication
104     """
105
106     # Assertion checks
107     assert A.shape[0] == A.shape[1], "Matrix A is not square!"
108     assert B.shape[0] == B.shape[1], "Matrix B is not square!"
109     assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"
110
111     # Perform multiplication, time it, & return result
112     time_start = time.time()
113     C = A @ B
114     cpu_time = 1000 * (time.time() - time_start)
115     print(f"CPU multiplication time: {cpu_time:.2f}ms")
116     return C
117
118
119 def main():
120     # Initialize argument parser
121     parser = argparse.ArgumentParser(
122         description="Compare naive matrix multi[lication between GPU and CPU"
123     )
124     parser.add_argument(
125         "--matrix_size",
126         type=int,
127         required=True,
128         help="Size of the square matrices being multiplied",
129     )

```

```

131     # Get required command line argument
132     args = parser.parse_args()
133     matrix_size = args.matrix_size
134
135     # Initialize CUDA driver
136     cuda.init()
137
138     # Get number of GPU's
139     num_gpus = cuda.Device.count()
140     print(f"Number of available GPUs: {num_gpus}")
141
142     # Prompt for Pacific ID
143     user_id = int(input("Please enter your Pacific ID: "))
144
145     # Determine which GPU to use
146     gpu_index = user_id % num_gpus
147     print(f"Using GPU index: {gpu_index}")
148     selected_gpu = cuda.Device(gpu_index)
149     print(f"Selected GPU: {selected_gpu.name()}")
150
151     # Create numpy square matrices of requested size
152     A = np.random.randn(matrix_size, matrix_size).astype(np.double)
153     B = np.random.randn(matrix_size, matrix_size).astype(np.double)
154
155     # Perform matrix multiplication on the CPU
156     cpu_result = matrix_mult_cpu(A, B)
157
158     # Perform GPU accelerated matrix multiplication
159     gpu_result = matrix_mult_gpu(A, B)
160
161     # Verify results are the same
162     if np.allclose(cpu_result, gpu_result):
163         print("Results are the same!")
164     else:
165         print("Results are different!")
166         print(f"CPU result: {cpu_result}")
167         print(f"GPU result: {gpu_result}")
168
169     if __name__ == "__main__":
170         main()

```

- 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.

```

1 import time
import numpy as np
3 import pycuda.autoinit
import pycuda.driver as cuda
5 from pycuda.compiler import SourceModule
import argparse

7
opt_kernel = SourceModule(
9     r"""
    #define TILEWIDTH 25 // MUST be the same as square thread block width/height!
11    __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
13    __shared__ double Ashared[TILEWIDTH][TILEWIDTH];
    __shared__ double Bshared[TILEWIDTH][TILEWIDTH];
15    int bx=blockIdx.x, by=blockIdx.y;
    int tx=threadIdx.x, ty=threadIdx.y;
17    int row=tx+bx*TILEWIDTH, col=ty+by*TILEWIDTH;
    double temp = 0; // Store element sums
19    // Loop over tiles Ashared and Bshared to compute matrix product elements in
        d_C
    for(int i = 0; i < width/TILEWIDTH; i++) {
21        // Collaboratively load correct elements into Ashared and Bshared to
            compute d_C
        Ashared[tx][ty] = d_A[i*TILEWIDTH + row*width + ty];
23        Bshared[tx][ty] = d_B[(i*TILEWIDTH+tx)*width + col];
        __syncthreads(); // Ensure all block threads catch up
25        // Loop over tiles to compute product entries
        for (int k = 0; k < TILEWIDTH; k++) {
27            temp += Ashared[tx][k] * Bshared[k][ty];
        }
29        __syncthreads(); // Let all block threads catch up
    }
31    d_C[row*width + col] = temp; // Insert element into d_C
    }
33    """
)
35

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

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

```

```

43
44
45 Returns:
46     np.ndarray: Result of matrix multiplication
47     """
48
49 # Assertion checks
50 assert A.shape[0] == A.shape[1], "Matrix A is not square!"
51 assert B.shape[0] == B.shape[1], "Matrix B is not square!"
52 assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"
53
54 matrix_size = A.shape[0]
55
56 # Send input array data to GPU device and time it
57 start_htod = time.time()
58 A_flatten = A.flatten()
59 A_gpu = cuda.mem_alloc(A_flatten.nbytes)
60 cudaMemcpy_htod(A_gpu, A_flatten)
61 B_flatten = B.flatten()
62 B_gpu = cuda.mem_alloc(B_flatten.nbytes)
63 cudaMemcpy_htod(B_gpu, B_flatten)
64 time_htod = (time.time() - start_htod) * 1000
65 print(f"GPU host-to-device transfer time: {time_htod:.2f}ms")
66
67 # Allocate space on device for output
68 C = np.empty_like(B).flatten()
69 C_gpu = cuda.mem_alloc(C.nbytes)
70
71 # Get device kernel and execute it
72 func = opt_kernel.get_function("opt_matrix_mult")
73 # Create 2D blocks of 32x32 threads (max 1024 threads allowed in block)
74 # Create grid of ceil(matrix_size // block_width) x ceil(matrix_size //
75     block_height) blocks
76 block_width = 25
77 block_height = 25
78 assert block_width * block_height <= 1024, "Max 1024 threads allowed in block!"
79 # Proper way to compute CEIL for grid dimensions
80 grid_width = (matrix_size + block_width - 1) // block_width
81 grid_height = (matrix_size + block_height - 1) // block_height
82 start_kernel = time.time()
83 func(
84     A_gpu,
85     B_gpu,
86     C_gpu,
87     np.int32(matrix_size),
88     block=(block_width, block_height, 1),
89     grid=(grid_width, grid_height),
90     shared=np.double().nbytes * block_width * block_height,
91 )
92 time_kernel = (time.time() - start_kernel) * 1000
93 print(f"GPU kernel execution time: {time_kernel:.2f}ms")
94
95 # Retrieve output data from the GPU and time it
96 start_dtoh = time.time()
97 cudaMemcpy_dtoh(C, C_gpu)

```

```

121     time_dtoh = (time.time() - start_dtoh) * 1000
122     print(f"GPU device-to-host transfer time: {time_dtoh:.2f}ms")
123
124     # Output total GPU time
125     total_time = time_htod + time_kernel + time_dtoh
126     print(f"Total GPU time: {total_time:.2f}ms")
127
128     # Return matrix multiplication result as properly shaped matrix
129     return C.reshape(matrix_size, matrix_size)
130
131 def matrix_mult_cpu(A: np.ndarray, B: np.ndarray):
132     """Perform Numpy matrix multiplication given the two np.ndarray matrices
133
134     Args:
135         A (np.ndarray): The first square matrix to multiply
136         B (np.ndarray): The second square matrix to multiply
137
138     Returns:
139         np.ndarray: Result of matrix multiplication
140     """
141
142     # Assertion checks
143     assert A.shape[0] == A.shape[1], "Matrix A is not square!"
144     assert B.shape[0] == B.shape[1], "Matrix B is not square!"
145     assert A.shape[1] == B.shape[0], "Matrices A and B cannot be multiplied!"
146
147     # Perform multiplication, time it, & return result
148     time_start = time.time()
149     C = A @ B
150     cpu_time = 1000 * (time.time() - time_start)
151     print(f"CPU multiplication time: {cpu_time:.2f}ms")
152     return C
153
154 def main():
155     # Initialize argument parser
156     parser = argparse.ArgumentParser(
157         description="Compare naive matrix multiplication between GPU and CPU"
158     )
159     parser.add_argument(
160         "--matrix_size",
161         type=int,
162         required=True,
163         help="Size of the square matrices being multiplied",
164     )
165
166     # Get required command line argument
167     args = parser.parse_args()
168     matrix_size = args.matrix_size
169
170     # Initialize CUDA driver
171     cuda.init()

```

```

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

153
155     # Prompt for Pacific ID
user_id = int(input("Please enter your Pacific ID: "))

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

163     # Create random numpy square matrices of requested size
# Sampled from normal distribution with mean of 3 and std deviation of 0.5
165     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)
167
169     # Perform matrix multiplication on the CPU
cpu_result = matrix_mult_cpu(A, B)

171     # Perform GPU accelerated matrix multiplication
gpu_result = opt_matrix_mult_gpu(A, B)
173
175     # Verify results are the same
if np.allclose(cpu_result, gpu_result):
    print("Results are the same!")
177 else:
    print("Results are different!")
    print(f"CPU result: {cpu_result}")
179     print(f"GPU result: {gpu_result}")
181
183 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

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