Writing Efficient Code and GPU Computing Homework

Please save your solutions as a **PDF** and upload it to Canvas.

Problem 1: Profiling and Vectorization

(a) Consider the following cProfile output from a data analysis program:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     5000    8.234    0.002    8.234    0.002 analysis.py:12(compute_distances)
     5000    0.089    0.000    0.089    0.000 analysis.py:28(normalize_vector)
  2500000    1.456    0.000    1.456    0.000 analysis.py:35(squared_diff)
        1    0.002    0.002    9.781    9.781 analysis.py:50(main)
```

Which function should you optimize first? Explain your reasoning based on the profiling data. What percentage of the total runtime does this function account for?

Answer: The function to optimize first is `compute_distances`, as it has the highest total time (`tottime`) of 8.234 seconds out of the total runtime of 9.781 seconds. This means it accounts for approximately 84.2% of the total runtime (calculated as (8.234 / 9.781) * 100). Optimizing this function would likely yield the most significant performance improvement for the overall program.

(b) The following function computes weighted squared differences between two arrays:

```
def weighted_squared_diff_loop(x, y, w):
    """Compute sum of weighted squared differences using a loop."""
    n = len(x)
    total = 0.0
    for i in range(n):
        diff = x[i] - y[i]
        total += w[i] * diff * diff
    return total
```

Write a vectorized version of this function using NumPy operations. Your function should produce the same result but without explicit Python loops. For example, `weighted_squared_diff(np.array([1, 2, 3]), np.array([0, 1, 1]), np.array([1, 2, 3]))` should return `15.0`.

```
import numpy as np

def weighted_squared_diff(x, y, w): return np.sum(w * (x - y)**2)

# Example usage:
result = weighted_squared_diff(np.array([1, 2, 3]), np.array([0, 1, 1]), np.array([1, 2, 3]))
print(result)  # Output: 15
```

```
15
```

(c) Write a function that transforms an array by replacing negative values with zero and scaling all positive values by their mean. For example, given `np.array([-2, 4, -1, 6, 2])`, the positive values are `[4, 6, 2]` with mean `4.0`, so the result should be `np.array([0, 1, 0, 1.5, 0.5])`. Use boolean indexing instead of loops.

```
import numpy as np
def transform_array(arr):
    arr = np.array(arr, dtype=float)
    pos_mask = arr > 0
    if np.any(pos_mask):
        mean_val = arr[pos_mask].mean()
        arr[pos_mask] = arr[pos_mask] / mean_val
    arr[~pos_mask] = 0
    return arr

# Example usage:
result = transform_array(np.array([-2, 4, -1, 6, 2]))
print(result) # [0, 1, 0, 1.5, 0.5]
```

```
[0.  1.  0.  1.5 0.5]
```

Problem 2: Parallelization and JIT Compilation

(a) The following function computes the mean of a bootstrap sample:

```
def compute_bootstrap_mean(args):
    """Compute mean of a bootstrap sample."""
    data, seed = args
    rng = np.random.RandomState(seed)
    sample = rng.choice(data, size=len(data), replace=True)
    return np.mean(sample)
```

Write a function that uses `multiprocessing.Pool` to compute `n_bootstrap` bootstrap means in parallel. Each bootstrap iteration should receive a unique seed to ensure different random samples. Return a list of the bootstrap means.

```
import numpy as np
import multiprocessing as mp

def compute_bootstrap_mean(args):
    """Compute mean of a bootstrap sample."""
    data, seed = args
    rng = np.random.RandomState(seed)
    sample = rng.choice(data, size=len(data), replace=True)
    return np.mean(sample)


def parallel_bootstrap(data, n_bootstrap, n_workers=4):
    """Compute bootstrap means in parallel."""
    seeds = range(n_bootstrap)
    tasks = [(data, seed) for seed in seeds]
    with mp.Pool(processes=n_workers) as pool:
        results = pool.map(compute_bootstrap_mean, tasks)
    return results

data = np.array([1, 2, 3, 4, 5])
bootstrap_means = parallel_bootstrap(data, n_bootstrap=10, n_workers=2)
print(bootstrap_means)  # Output: List of bootstrap means
```

```
[np.float64(3.6), np.float64(3.2), np.float64(2.6), np.float64(2.2), np.float64(2.0), np.float64(2.6), np.float64(3.4
```

(b) Write a Numba-optimized function that computes the running maximum of an array. For each position `i`, the output should contain the maximum of all elements from index 0 to `i` (inclusive). For example, `running_max(np.array([3, 1, 4, 1, 5, 9, 2, 6]))` should return `np.array([3, 3, 4, 4, 5, 9, 9, 9])`.

```
from numba import njit
import numpy as np

@njit
def running_max(arr):
    """Compute running maximum of array."""
    n = len(arr)
    result = np.empty(n, dtype=arr.dtype)
    current_max = arr[0]
    for i in range(n):
        if arr[i] > current_max:
            current_max = arr[i]
        result[i] = current_max
    return result

# Example usage:
result = running_max(np.array([1, 3, 2, 5, 4]))
print(result)  # Output: [1 3 3 5 5]
```

```
[1 3 3 5 5]
```

(c) The following Numba function attempts to filter an array to keep only positive values, but it fails to compile. Explain why it fails and provide a corrected version that compiles successfully with `@njit`.

```
from numba import njit
import numpy as np


@njit
def filter_positive_broken(arr):
    """Return array containing only positive values (BROKEN)."""
    result = []
    for x in arr:
        if x > 0:
            result.append(x)
    return np.array(result)
```

When define result = [], Numba doesn't know what type of data goes inside the list. In compiled machine code, variables must have strict, known types. Also, we generally need to know the size of an array before you write to it to allocate the correct amount of memory. Here's a corrected version that pre-allocates an array for the positive values:

```
from numba import njit
import numpy as np
@njit
def filter_positive_fixed(arr):
    count = 0
    for value in arr:
        if value > 0:
            count += 1
    result = np.empty(count, dtype=arr.dtype)
    index = 0
    for value in arr:
        if value > 0:
            result[index] = value
            index += 1
    return result

## Example usage:
data = np.array([-5, 3, -1, 4, 0, 2])
positive_data = filter_positive_fixed(data)
print(positive_data)  # Output: [3 4 2]
```

```
[3 4 2]
```

Problem 3: GPU Computing Fundamentals

(a) For each of the following computational tasks, state whether it would benefit from GPU acceleration and explain why or why not.

1. Computing the mean of 500 numbers
2. Multiplying two 5000x5000 matrices
3. Reading a 10GB CSV file from disk
4. Running 1 million independent Monte Carlo simulations
5. Computing Fibonacci numbers recursively

Answer:

1. No, this task is too small to benefit from GPU acceleration due to the overhead of data transfer between CPU and GPU.
2. Yes, this task is highly parallelizable and involves a large amount of computation, making it suitable for GPU acceleration.
3. No, this is an I/O-bound task rather than a compute-bound task, so GPU acceleration would not help.
4. Yes, this task is embarrassingly parallel and can benefit from GPU acceleration as each simulation can be run independently.
5. No, this task is inherently sequential and does not benefit from parallelization.

(b) The following code runs slowly despite using GPU. Identify the performance problem and rewrite the code to fix it. The goal is to compute the sum of squares for 1000 different arrays.

```
import cupy as cp
import numpy as np

results = []
for i in range(1000):
    data = np.random.randn(10000)  # Generate on CPU
    gpu_data = cp.asarray(data)    # Transfer to GPU
    result = cp.sum(gpu_data ** 2) # Compute on GPU
    results.append(result.get())   # Transfer back to CPU
```

```
print(f"Total: {sum(results)}")
```

Ans: The performance problem is excessive data transfer between the CPU and GPU. Inside the loop, the codes move data from Host (CPU) to Device (GPU) and back 1000 times, which is inefficient. Also, by generating random numbers on the CPU, the codes are forcing the high-speed GPU to wait for the slower CPU.

Write an efficient version that minimizes data transfers between CPU and GPU.

```
import cupy as cp
import numpy as np

all_data_gpu = cp.random.randn(1000, 10000)
total_sum_gpu = cp.sum(all_data_gpu ** 2)
total_result = total_sum_gpu.get()
print(f"Total: {total_result}")
```

```
Total: 9992999.446989737
```

Problem 4: CuPy and PyTorch

(a) Convert the following NumPy code to CuPy. The function computes z-score normalization and then the correlation matrix.

```
import numpy as np


def correlation_matrix_numpy(X):
    """Compute correlation matrix after z-score normalization.

    X has shape (n_samples, n_features).
    """
    # Z-score normalize each column
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    Z = (X - mean) / std

    # Compute correlation matrix
    n = X.shape[0]
    corr = (Z.T @ Z) / n
    return corr
```

Write the CuPy version that performs the computation on GPU and returns the result as a NumPy array.

```
import cupy as cp
import numpy as np

def correlation_matrix_cupy(X):
    X_gpu = cp.asarray(X)
    mean_gpu = cp.mean(X_gpu, axis=0)
    std_gpu = cp.std(X_gpu, axis=0)
    Z_gpu = (X_gpu - mean_gpu) / std_gpu
    n = X_gpu.shape[0]
    corr_gpu = (Z_gpu.T @ Z_gpu) / n
    return corr_gpu.get()

# Example usage:
X_input = np.array([
    [1.0, 2.0, 3.0],
    [2.0, 3.0, 4.0],
    [3.0, 4.0, 5.0],
    [4.0, 5.0, 6.0],
    [5.0, 6.0, 1.0],  # Breaking the perfect correlation here
    [6.0, 1.0, 2.0],
    [7.0, 2.0, 3.0],
    [8.0, 3.0, 4.0],
    [9.0, 4.0, 5.0],
    [1.0, 5.0, 6.0]
])

correlation_matrix = correlation_matrix_cupy(X_input)
```

```
print("Input Shape:", X_input.shape)
print("Correlation Matrix Shape:", correlation_matrix.shape)
print("\nComputed Correlation Matrix:")
print(np.round(correlation_matrix, 4))
```

```
Input Shape: (10, 3)
Correlation Matrix Shape: (3, 3)

Computed Correlation Matrix:
[[ 1.     -0.1466 -0.1952]
 [-0.1466  1.      0.3169]
 [-0.1952  0.3169  1.    ]]
```

(b) The following PyTorch code has a bug that causes a runtime error. Identify the error and provide the corrected code.

```
import torch
import numpy as np


def process_data(numpy_array):
    """Process data using PyTorch on GPU."""
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Convert to tensor and move to GPU
    x = torch.from_numpy(numpy_array).to(device)

    # Create another tensor for computation
    weights = torch.ones(len(numpy_array))

    # Weighted sum
    result = torch.sum(x * weights)

    return result.item()
```

Ans: The Error: The code fails because it tries to multiply two tensors located on different devices (CPU and GPU): x is moved to the GPU while weights is created on the CPU by default. Here is the corrected PyTorch code that ensures both tensors are on the same device:

```
import torch
def process_data(numpy_array):
    if torch.cuda.is_available():
        device = torch.device('cuda')
    elif torch.backends.mps.is_available():
        device = torch.device('mps')
    else:
        device = torch.device('cpu')

    x = torch.from_numpy(numpy_array).to(device)
    weights = torch.ones(len(numpy_array), device=device)
    result = torch.sum(x * weights)

    return result.item()

# Example usage:
data = np.array([10.0, 20.0, 30.0, 40.0], dtype=np.float32)
output = process_data(data)
print("-" * 30)
print(f"Input Data: {data}")
print(f"Output Sum: {output}")
```

```
------------------------------
Input Data: [10. 20. 30. 40.]
Output Sum: 100.0
```

(c) Explain why the following GPU timing code gives incorrect measurements. Then provide corrected code that accurately measures GPU computation time.

```
import torch
import time

device = torch.device('cuda')
a = torch.randn(5000, 5000, device=device)
b = torch.randn(5000, 5000, device=device)
```

```
start = time.perf_counter()
c = torch.mm(a, b)
elapsed = time.perf_counter() - start
print(f"Time: {elapsed*1000:.2f} ms")
```

Ans: The timing code gives incorrect measurements because GPU operations are asynchronous. When you call a GPU operation, it is queued and may not complete immediately. Therefore, the time measured does not account for the actual computation time on the GPU.

```
import torch
import time

device = torch.device('cuda')
a = torch.randn(5000, 5000, device=device)
b = torch.randn(5000, 5000, device=device)
torch.cuda.synchronize()
start = time.perf_counter()
c = torch.mm(a, b)
torch.cuda.synchronize()
elapsed = time.perf_counter() - start
print(f"Time: {elapsed*1000:.2f} ms")
```

```
Time: 117.32 ms
```

Problem 5: Performance Comparison

(a) In extreme value statistics, we often need to estimate the probability that the maximum of n independent standard normal random variables exceeds a threshold t. This can be done via Monte Carlo simulation: generate n normal values, take the maximum, and check if it exceeds t. Repeat this many times and compute the proportion that exceed t.

Implement two versions of this simulation:

1. A Numba-optimized CPU version using `@njit`
2. A CuPy GPU version using vectorized operations

Both functions should take parameters `n` (number of normal values per trial), `t` (threshold), and `n_simulations` (number of Monte Carlo trials), and return the estimated probability.

```
from numba import njit
import numpy as np
import cupy as cp

@njit
def estimate_prob_numba(n, t, n_simulations):
    count = 0
    for i in range(n_simulations):
        current_max = -np.inf
        for j in range(n):
            val = np.random.randn()
            if val > current_max:
                current_max = val

        if current_max > t:
            count += 1

    return count / n_simulations

def estimate_prob_cupy(n, t, n_simulations):
    data = cp.random.randn(n_simulations, n)
    max_vals = cp.max(data, axis=1)
    probability = cp.mean(max_vals > t)
    return probability.item()
```

(b) Design an experiment to find the "crossover point" where the GPU version becomes faster than the CPU version. Your experiment should vary the problem size (e.g., `n_simulations`) and measure execution time for both implementations. Describe what factors affect where this crossover occurs and what values you would test.

```
import time
import numpy as np
import cupy as cp
import pandas as pd
```

```
def benchmark_crossover():
    n = 100
    t = 2.0
    simulation_counts = [1_000, 10_000, 100_000, 1_000_000, 10_000_000]
    results = []

    print(f"{'Simulations':<15} | {'Numba (CPU)':<15} | {'CuPy (GPU)':<15} | {'Winner':<10}")
    print("-" * 65)

    estimate_prob_numba(10, t, 10)
    estimate_prob_cupy(10, t, 10)

    for n_sim in simulation_counts:
        start_cpu = time.perf_counter()
        estimate_prob_numba(n, t, n_sim)
        end_cpu = time.perf_counter()
        time_cpu = end_cpu - start_cpu

        cp.get_default_memory_pool().free_all_blocks()
        cp.cuda.Device().synchronize()
        start_gpu = time.perf_counter()

        estimate_prob_cupy(n, t, n_sim)

        cp.cuda.Device().synchronize()
        end_gpu = time.perf_counter()
        time_gpu = end_gpu - start_gpu

        winner = "CPU" if time_cpu < time_gpu else "GPU"
        results.append({
            "n_simulations": n_sim,
            "CPU_Time": time_cpu,
            "GPU_Time": time_gpu,
            "Winner": winner
        })

        print(f"{n_sim:<15} | {time_cpu:.6f}s | {time_gpu:.6f}s | {winner}")

    return pd.DataFrame(results)

df_results = benchmark_crossover()
print(df_results.to_markdown(index=False))
```

```
Simulations     | Numba (CPU)     | CuPy (GPU)      | Winner
-----------------------------------------------------------------
1000            | 0.002491s | 2.618061s | CPU
10000           | 0.024998s | 2.636630s | CPU
100000          | 0.249434s | 0.017919s | GPU
1000000         | 2.523152s | 0.176892s | GPU
10000000        | 27.504516s | 0.818189s | GPU
|   n_simulations |    CPU_Time |   GPU_Time | Winner   |
|----------------:|------------:|-----------:|:---------|
|            1000 |  0.00249071 |  2.61806   | CPU      |
|           10000 |  0.0249984  |  2.63663   | CPU      |
|          100000 |  0.249434   |  0.0179187 | GPU      |
|         1000000 |  2.52315    |  0.176892  | GPU      |
|        10000000 | 27.5045     |  0.818189  | GPU      |
```

Factors Affecting the Crossover Point:

1. Kernel Launch Overhead: The time taken to launch GPU kernels can be significant for small problem sizes, making the CPU faster for smaller workloads.
2. Core Saturation: A GPU has thousands of cores (e.g., 2,000–10,000). To be efficient, it needs enough work to keep all of them busy simultaneously.
3. Memory Transfer vs. Compute Ratio: If the system runs out of GPU memory (VRAM), it might crash or slow down. The Numba version uses almost no memory.

Test values: With the fixed values n = 100, t = 2, run the Test values in Logarithmic scale: [1_000, 10000, 100000, 1000000, 10000000].

(c) Suppose you need to run a very large simulation with `n_simulations = 100_000_000` but your GPU only has 8GB of memory. The naive CuPy implementation would require generating a matrix of shape `(n_simulations, n)` which may not fit in memory. Write a

batched version that processes the simulations in chunks to stay within memory limits.

```
import cupy as cp

def estimate_prob_cupy_batched(n, t, n_simulations, batch_size=1000000):
    total_successes = 0
    for i in range(0, n_simulations, batch_size):
        current_batch_size = min(batch_size, n_simulations - i)
        batch_data = cp.random.randn(current_batch_size, n)
        batch_maxs = cp.max(batch_data, axis=1)
        count = cp.sum(batch_maxs > t).item()
        total_successes += count
    return total_successes / n_simulations

# Example usage:
n_vars = 100
threshold = 3.0
total_sims = 20_000_000
batch_sz = 2_000_000

print(f"Running {total_sims:,} simulations on GPU...")
print(f"Batch size: {batch_sz:,}")

start_time = time.perf_counter()
probability = estimate_prob_cupy_batched(n_vars, threshold, total_sims, batch_size=batch_sz)
cp.cuda.Device().synchronize()
end_time = time.perf_counter()

print("-" * 40)
print(f"Estimated Probability: {probability:.6f} ({probability*100:.2f}%)")
print(f"Execution Time:        {end_time - start_time:.4f} seconds")
```

```
Running 20,000,000 simulations on GPU...
Batch size: 2,000,000
----------------------------------------
Estimated Probability: 0.126487 (12.65%)
Execution Time:        4.0032 seconds
```