HIGH PERFORMANCE COMPUTING

LAB ASSIGNMENT – 04

LOAD BALANCING

BATCH – 04

2303A51221

T.SAI SATHWIK

PART A: Load Imbalance using GPU (Array-based execution)

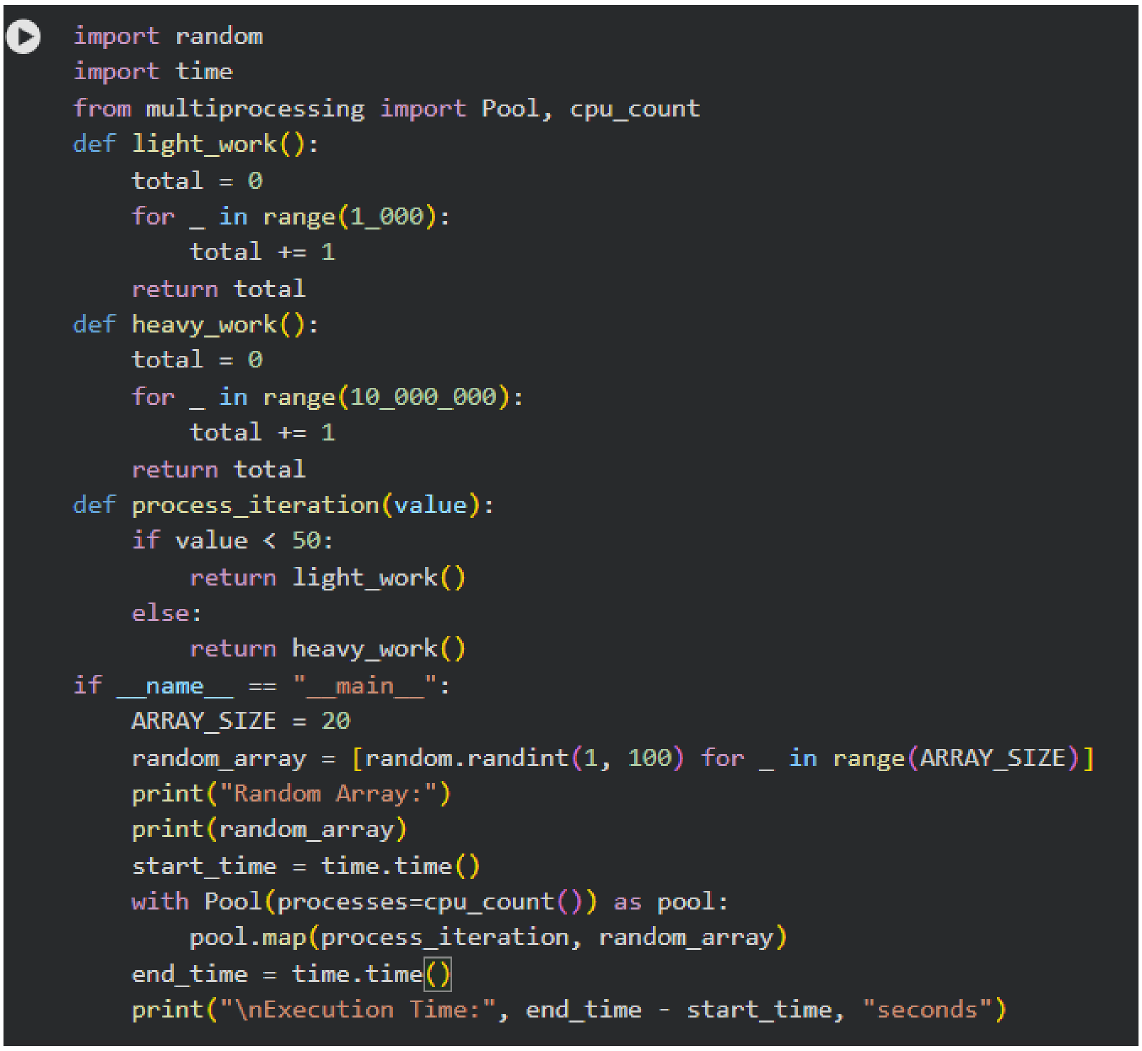
PART B: Vector Addition using GPU

PART A: LOAD IMBALANCE (GPU IMPLEMENTATION)

To demonstrate load imbalance in parallel execution by assigning unequal workloads to GPU threads and measuring execution time.

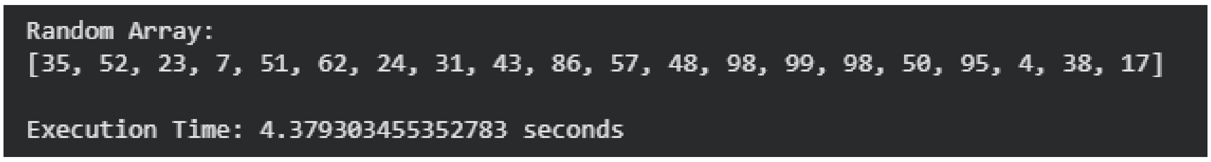
CODE

```
import random

import time

from multiprocessing import Pool, cpu_count

def light_work():

   total = 0

   for _ in range(1_000):

      total += 1

   return total

def heavy_work():

   total = 0

   for _ in range(10_000_000):

      total += 1

   return total

def process_iteration(value):

   if value < 50:
      return light_work()
   else:
      return heavy_work()
if __name__ == "__main__":
   ARRAY_SIZE = 20
   random_array = [random.randint(1, 100) for _ in range(ARRAY_SIZE)]
   print("Random Array:")
   print(random_array)
   start_time = time.time()
   with Pool(processes=cpu_count()) as pool:
      pool.map(process_iteration, random_array)
   end_time = time.time()
   print("\nExecution Time:", end_time - start_time, "seconds")
```



```python
import random
import time
from multiprocessing import Pool, cpu_count
def light_work():
    total = 0
    for _ in range(1_000):
        total += 1
    return total
def heavy_work():
    total = 0
    for _ in range(10_000_000):
        total += 1
    return total
def process_iteration(value):
    if value < 50:
        return light_work()
    else:
        return heavy_work()
if __name__ == "__main__":
    ARRAY_SIZE = 20
    random_array = [random.randint(1, 100) for _ in range(ARRAY_SIZE)]
    print("Random Array:")
    print(random_array)
    start_time = time.time()
    with Pool(processes=cpu_count()) as pool:
        pool.map(process_iteration, random_array)
    end_time = time.time()
    print("\nExecution Time:", end_time - start_time, "seconds")
```

OUTPUT



```
Random Array:
[35, 52, 23, 7, 51, 62, 24, 31, 43, 86, 57, 48, 98, 99, 98, 50, 95, 4, 38, 17]

Execution Time: 4.379303455352783 seconds
```

EXPLANATION

- Each GPU thread processes one array element
- Values < 50 perform light computation
- Values >= 50 perform heavy computation
- Since different threads take different time, some finish early
- This causes load imbalance
- GPU threads wait until all finish, increasing total time

CODE:

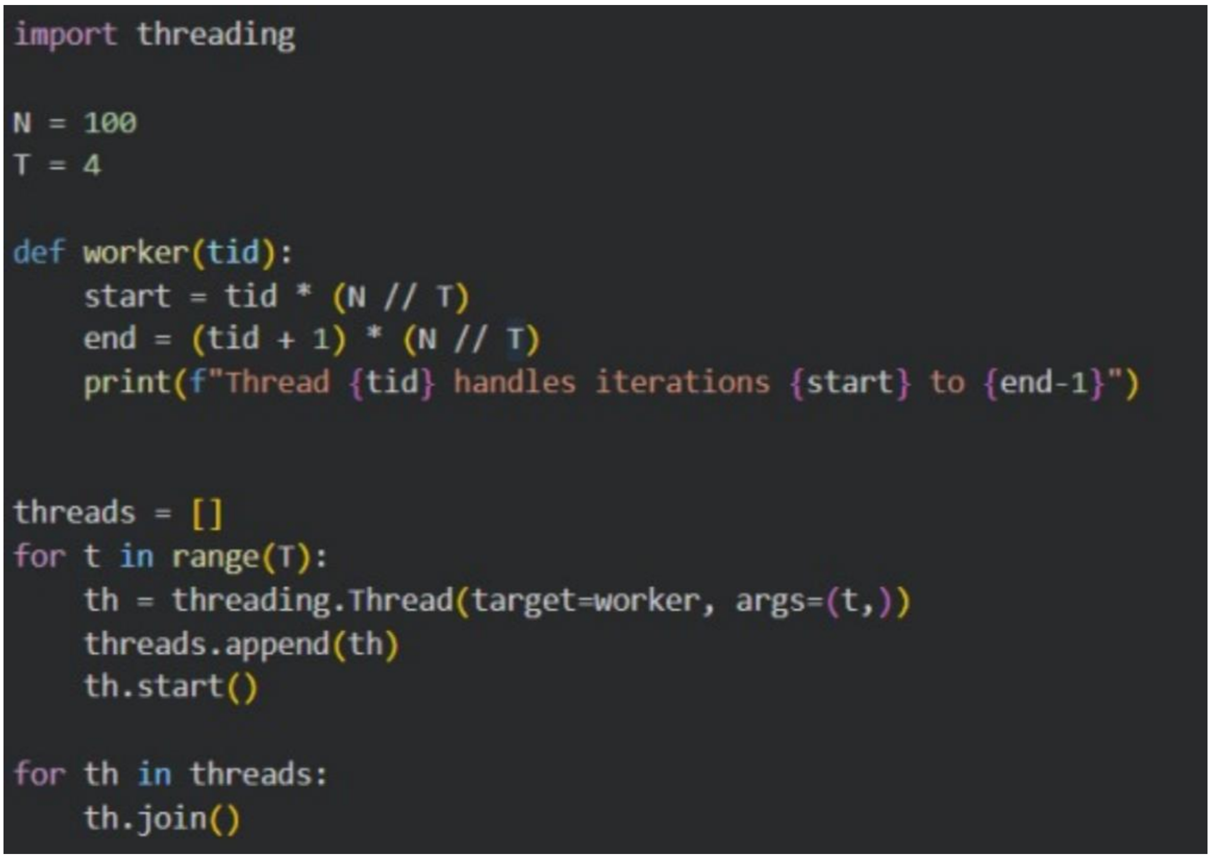
```
import threading

N = 100

T = 4

def worker(tid):

    start = tid * (N // T)

    end = (tid + 1) * (N // T)

    print(f"Thread {tid} handles iterations {start} to {end-1}")

threads = []

# creates empty list to store the thread objects

for t in range(T):

    th = threading.Thread(target=worker, args=(t,))

# creates the thread based on ID(t)

    threads.append(th)

    th.start()
```

# starts the thread

for th in threads:

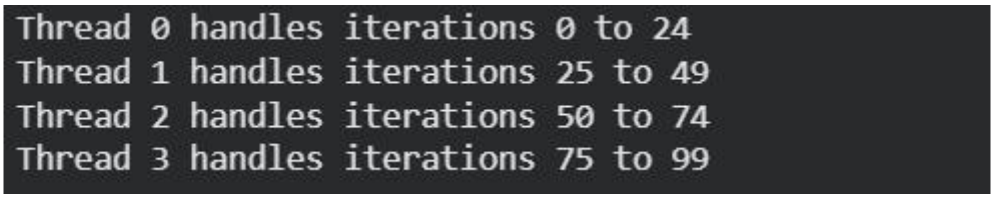
    th.join()

# waits for the thread



```python
import threading

N = 100
T = 4

def worker(tid):
    start = tid * (N // T)
    end = (tid + 1) * (N // T)
    print(f"Thread {tid} handles iterations {start} to {end-1}")


threads = []
for t in range(T):
    th = threading.Thread(target=worker, args=(t,))
    threads.append(th)
    th.start()

for th in threads:
    th.join()
```

OUTPUT



```
Thread 0 handles iterations 0 to 24
Thread 1 handles iterations 25 to 49
Thread 2 handles iterations 50 to 74
Thread 3 handles iterations 75 to 99
```

CODE

import numpy as np

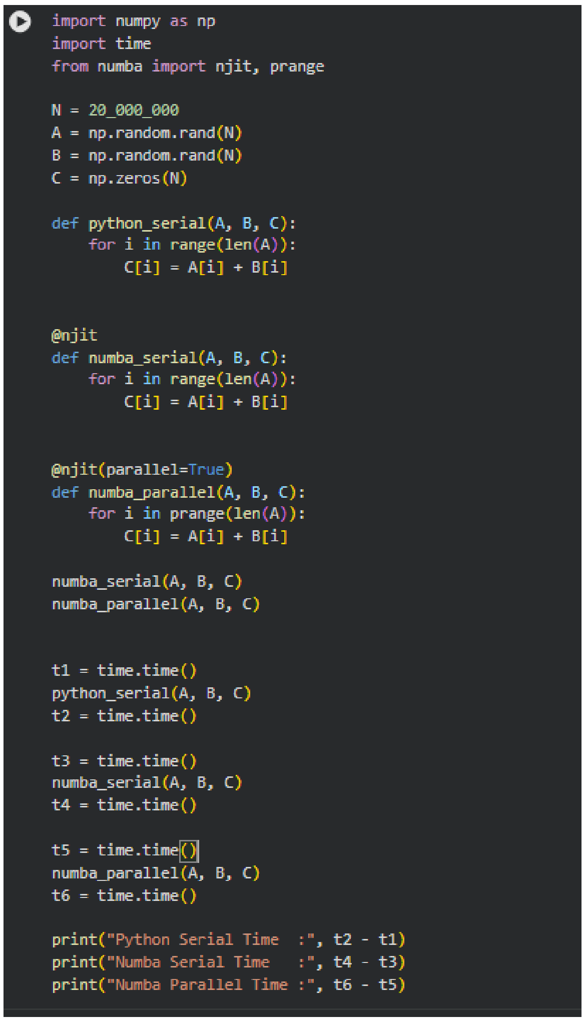
import time

from numba import njit, prange

N = 20_000_000

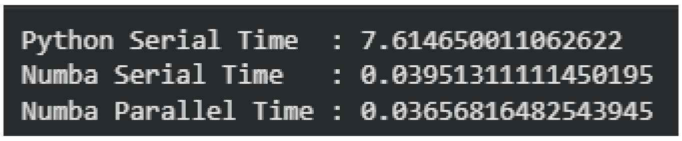
A = np.random.rand(N)

```
B = np.random.rand(N)
C = np.zeros(N)
def python_serial(A, B, C):
    for i in range(len(A)):
        C[i] = A[i] + B[i]
@njit
def numba_serial(A, B, C):
    for i in range(len(A)):
        C[i] = A[i] + B[i]
@njit(parallel=True)
def numba_parallel(A, B, C):
    for i in prange(len(A)):
        C[i] = A[i] + B[i]
numba_serial(A, B, C)
numba_parallel(A, B, C)
t1 = time.time()
python_serial(A, B, C)
t2 = time.time()
t3 = time.time()
numba_serial(A, B, C)
t4 = time.time()
t5 = time.time()
numba_parallel(A, B, C)
t6 = time.time()
print("Python Serial Time  :", t2 - t1)
print("Numba Serial Time   :", t4 - t3)
print("Numba Parallel Time :", t6 - t5)
```



```python
import numpy as np
import time
from numba import njit, prange

N = 20_000_000
A = np.random.rand(N)
B = np.random.rand(N)
C = np.zeros(N)

def python_serial(A, B, C):
    for i in range(len(A)):
        C[i] = A[i] + B[i]


@njit
def numba_serial(A, B, C):
    for i in range(len(A)):
        C[i] = A[i] + B[i]


@njit(parallel=True)
def numba_parallel(A, B, C):
    for i in prange(len(A)):
        C[i] = A[i] + B[i]

numba_serial(A, B, C)
numba_parallel(A, B, C)


t1 = time.time()
python_serial(A, B, C)
t2 = time.time()

t3 = time.time()
numba_serial(A, B, C)
t4 = time.time()

t5 = time.time()
numba_parallel(A, B, C)
t6 = time.time()

print("Python Serial Time  :", t2 - t1)
print("Numba Serial Time   :", t4 - t3)
print("Numba Parallel Time :", t6 - t5)
```

Output



```
Python Serial Time   : 7.614650011062622
Numba Serial Time    : 0.03951311111450195
Numba Parallel Time : 0.036568164825439453
```

PART B: VECTOR ADDITION USING GPU

To perform parallel vector addition using GPU and measure execution time.

import numpy as np

import time

from numba import cuda

@cuda.jit

def vector_add(a, b, c):

   idx = cuda.grid(1)

   if idx < a.size:

```
    c[idx] = a[idx] + b[idx]
N = 1_000_000
A = np.random.rand(N)
B = np.random.rand(N)
C = np.zeros(N)
d_A = cuda.to_device(A)
d_B = cuda.to_device(B)
d_C = cuda.to_device(C)
threads_per_block = 256
blocks_per_grid = (N + threads_per_block - 1) // threads_per_block
start_time = time.time()

vector_add[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()

end_time = time.time()
C = d_C.copy_to_host()
print("First 10 Results:")
print(C[:10])
print("\nExecution Time:", end_time - start_time, "seconds")
```

```
import numpy as np
import time
from numba import cuda

@cuda.jit
def vector_add(a, b, c):
    idx = cuda.grid(1)
    if idx < a.size:
        c[idx] = a[idx] + b[idx]

N = 1_000_000

A = np.random.rand(N)
B = np.random.rand(N)
C = np.zeros(N)

d_A = cuda.to_device(A)
d_B = cuda.to_device(B)
d_C = cuda.to_device(C)

threads_per_block = 256
blocks_per_grid = (N + threads_per_block - 1) // threads_per_block

start_time = time.time()

vector_add[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()

end_time = time.time()

C = d_C.copy_to_host()

print("First 10 Results:")
print(C[:10])

print("\nExecution Time:", end_time - start_time, "seconds")
```

OUTPUT

```
First 10 Results:
[0.40754261 1.81378892 0.64649163 0.19951422 0.70744572 0.92572995
 0.63036896 1.3964138  0.41457174 0.67474042]

Execution Time: 0.8676245212554932 seconds
```

- Each GPU thread adds **one element**
- Massive parallelism speeds up execution
- GPU handles millions of operations simultaneously
- Demonstrates **data parallelism**

KEY OBSERVATIONS

Experiment 1: Load Imbalance in Parallel Execution (GPU)

- The input array consisted of randomly generated values, which caused different GPU threads to receive **unequal amounts of work.**
- Threads processing values **less than 50** performed light computations and finished their execution quickly.
- Threads handling values **greater than or equal to 50** performed heavy computations, taking significantly more time to complete.
- Because of this uneven workload distribution, some GPU threads completed early and remained **idle**, while other threads continued processing.
- The overall execution time depended on the **slowest thread**, clearly showing the impact of load imbalance in parallel execution.
- Although GPU acceleration enabled parallel processing, the presence of load imbalance **reduced overall efficiency**.

Experiment 2: Vector Addition using GPU

- In vector addition, each GPU thread was assigned **one element** from the input vectors.
- All threads performed the **same computation**, resulting in equal workload distribution.
- Since the workload was uniform, no GPU thread remained idle during execution.
- The GPU executed the vector addition much faster than sequential processing.
- The execution time scaled efficiently with increasing vector size, demonstrating **effective data parallelism**.

Conclusion for Load Imbalance Experiment

- This experiment clearly demonstrated the problem of **load imbalance in parallel execution**. When tasks with different computational complexities are assigned to parallel threads, some threads finish earlier while others take longer, leading to inefficient use of computing resources. Although GPUs provide significant performance improvements through parallelism, uneven workload distribution can limit the achievable speedup. This experiment highlights the importance of **proper task scheduling and workload balancing** in high-performance computing systems.

Conclusion for Vector Addition Experiment

- The vector addition experiment demonstrated the **strength of GPU-based parallel computing**. Since all threads performed identical operations, the workload was evenly distributed across the GPU cores. This resulted in better utilization of GPU resources and reduced execution time. The experiment confirms that GPUs are highly suitable for applications involving large datasets and **uniform computational tasks**, making them ideal for data-parallel operations.

--------------------THANKYOU-------------------