1)

import time

import random

from concurrent.futures import ThreadPoolExecutor

# Mock functions to simulate ML tasks' workload

def simulate\_cnn(dataset):

time.sleep(random.uniform(0.5, 1.5)) # Simulating variable computation time

def simulate\_mlp(dataset):

time.sleep(random.uniform(0.2, 0.7))

def simulate\_rnn(dataset):

time.sleep(random.uniform(0.4, 1.0))

def simulate\_linear\_regression(dataset):

time.sleep(random.uniform(0.1, 0.3))

def simulate\_logistic\_regression(dataset):

time.sleep(random.uniform(0.2, 0.6))

def simulate\_svm(dataset):

time.sleep(random.uniform(0.3, 0.8))

# Function to simulate dataset loading

def load\_dataset(name):

# Mock dataset loading time

time.sleep(random.uniform(0.1, 0.2))

return f"{name}\_data"

# Performance measurement simulation

def measure\_performance(task\_func, dataset, parallel=False):

start\_time = time.time()

if parallel:

with ThreadPoolExecutor(max\_workers=2) as executor:

executor.submit(task\_func, dataset)

else:

task\_func(dataset)

end\_time = time.time()

return end\_time - start\_time

# Example usage

datasets = ['VGGFace2', 'NIST', 'CIFAR-10', 'SYNTHETIC', 'MNIST']

ml\_tasks = [simulate\_cnn, simulate\_mlp, simulate\_rnn, simulate\_linear\_regression, simulate\_logistic\_regression, simulate\_svm]

for dataset in datasets:

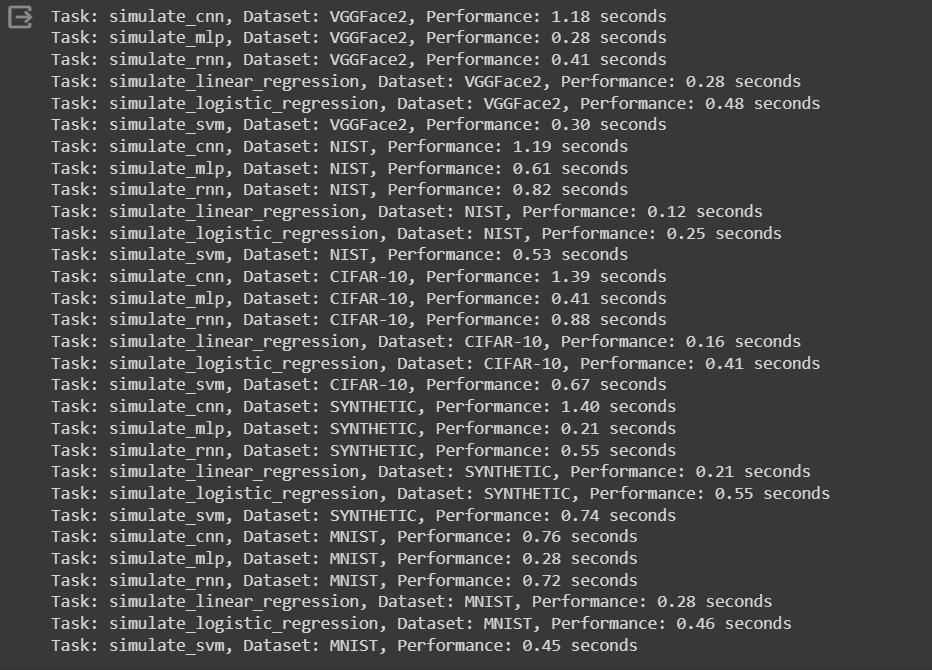
data = load\_dataset(dataset)

for task in ml\_tasks:

performance = measure\_performance(task, data, parallel=True if dataset == 'SYNTHETIC' else False)

print(f"Task: {task.\_name\_}, Dataset: {dataset}, Performance: {performance:.2f} seconds")

**OUTPUT:**



2)

import time

import random

from concurrent.futures import ThreadPoolExecutor

import matplotlib.pyplot as plt

# Mock functions to simulate ML tasks' workload

def simulate\_cnn(dataset):

time.sleep(random.uniform(0.5, 1.5)) # Simulating variable computation time

def simulate\_mlp(dataset):

time.sleep(random.uniform(0.2, 0.7))

def simulate\_rnn(dataset):

time.sleep(random.uniform(0.4, 1.0))

def simulate\_linear\_regression(dataset):

time.sleep(random.uniform(0.1, 0.3))

def simulate\_logistic\_regression(dataset):

time.sleep(random.uniform(0.2, 0.6))

def simulate\_svm(dataset):

time.sleep(random.uniform(0.3, 0.8))

# Function to simulate dataset loading

def load\_dataset(name):

# Mock dataset loading time

time.sleep(random.uniform(0.1, 0.2))

return f"{name}\_data"

# Performance measurement simulation

def measure\_performance(task\_func, dataset, parallel=False):

start\_time = time.time()

if parallel:

with ThreadPoolExecutor(max\_workers=2) as executor:

executor.submit(task\_func, dataset)

else:

task\_func(dataset)

end\_time = time.time()

return end\_time - start\_time

# Example usage

datasets = ['VGGFace2', 'NIST', 'CIFAR-10', 'SYNTHETIC', 'MNIST']

ml\_tasks = [simulate\_cnn, simulate\_mlp, simulate\_rnn, simulate\_linear\_regression, simulate\_logistic\_regression, simulate\_svm]

results = {} # Dictionary to store results

for dataset in datasets:

data = load\_dataset(dataset)

results[dataset] = {} # Nested dictionary to store task performances for each dataset

for task in ml\_tasks:

performance = measure\_performance(task, data, parallel=True if dataset == 'SYNTHETIC' else False)

results[dataset][task.\_name\_] = performance

# Plotting

fig, ax = plt.subplots(figsize=(12, 8))

for dataset in datasets:

task\_performances = results[dataset]

tasks = list(task\_performances.keys())

performances = list(task\_performances.values())

ax.bar(tasks, performances, label=dataset)

ax.set\_xlabel('ML Task')

ax.set\_ylabel('Performance (seconds)')

ax.set\_title('Performance of ML Tasks on Different Datasets')

ax.legend()

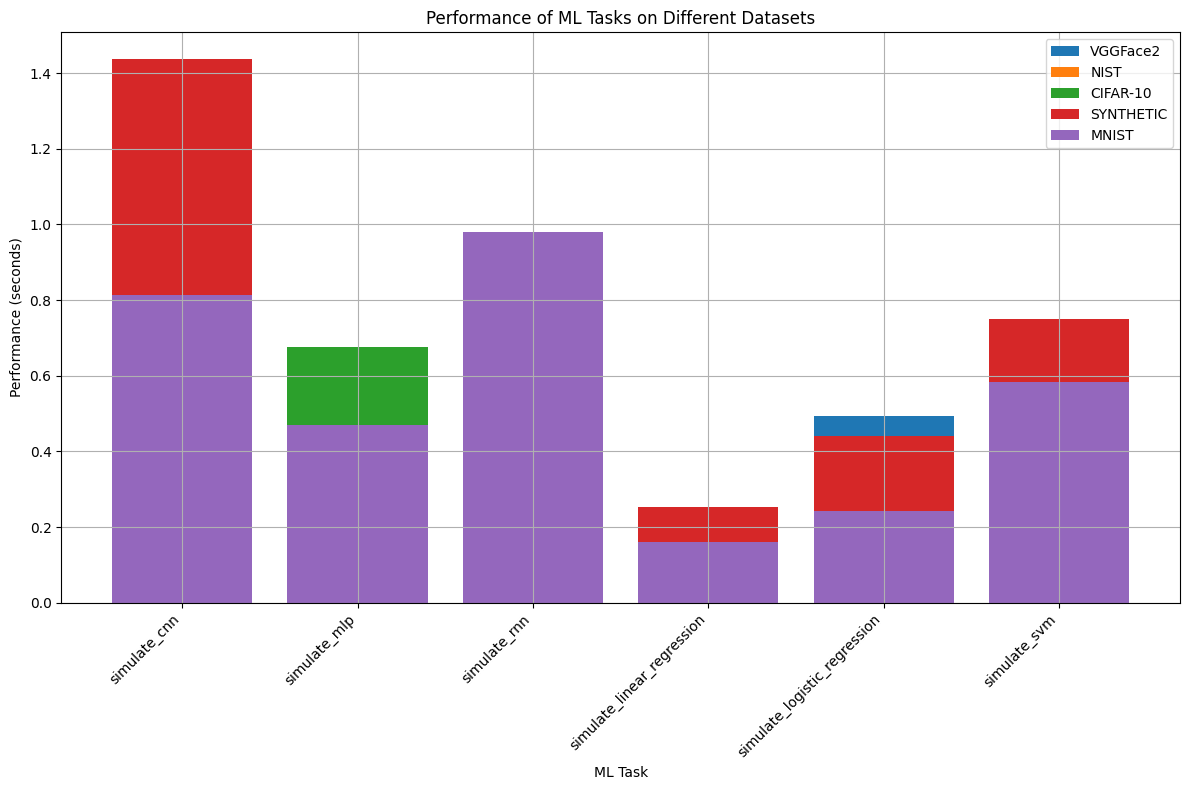
ax.grid(True)

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

**OUTPUT:**



3)

# Import necessary libraries

import numpy as np

import threading

from numba import cuda, float32

# Thread-safe random number generation function

def thread\_safe\_random(start, end):

rng = np.random.default\_rng() # Each thread gets its own RNG instance

return rng.integers(start, end)

def worker():

# Example worker function to demonstrate thread-safe RNG

print(f"Thread-safe random number: {thread\_safe\_random(1, 100)}")

# Efficient matrix operations using numpy

def efficient\_matrix\_operations():

A = np.random.rand(1000, 1000)

B = np.random.rand(1000, 1000)

# Matrix addition and subtraction

addition\_result = A + B

subtraction\_result = A - B

print("Matrix addition and subtraction performed efficiently.")

# GPU-accelerated matrix addition using Numba

@cuda.jit

def add\_matrices\_gpu(A, B, result):

x, y = cuda.grid(2)

if x < result.shape[0] and y < result.shape[1]:

result[x, y] = A[x, y] + B[x, y]

def gpu\_matrix\_addition():

A = np.random.rand(1000, 1000).astype(np.float32)

B = np.random.rand(1000, 1000).astype(np.float32)

result = np.zeros\_like(A)

# Define grid size for GPU execution

threadsperblock = (16, 16)

blockspergrid\_x = int(np.ceil(A.shape[0] / threadsperblock[0]))

blockspergrid\_y = int(np.ceil(A.shape[1] / threadsperblock[1]))

blockspergrid = (blockspergrid\_x, blockspergrid\_y)

# Perform the matrix addition on GPU

add\_matrices\_gpu[blockspergrid, threadsperblock](A, B, result)

print("Matrix addition performed on GPU.")

# Main execution block to run the examples

if \_name\_ == "\_main\_":

# Demonstrate thread-safe RNG with 5 worker threads

threads = [threading.Thread(target=worker) for \_ in range(5)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

# Perform efficient matrix operations

efficient\_matrix\_operations()

# Demonstrate GPU matrix addition if CUDA is available

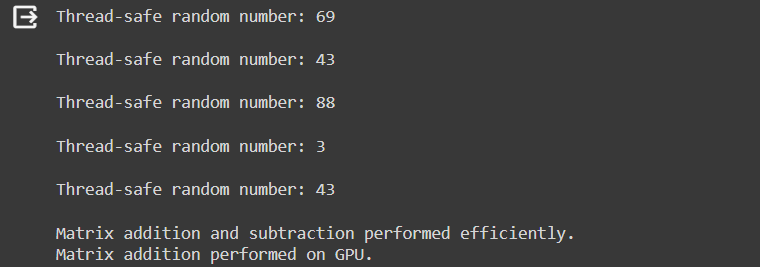
if cuda.is\_available():

gpu\_matrix\_addition()

else:

print("CUDA is not available. GPU matrix addition skipped.")

**OUPUT:**



4)

import numpy as np

import threading

from numba import cuda, float32

import matplotlib.pyplot as plt

# Thread-safe random number generation function

def thread\_safe\_random(start, end):

rng = np.random.default\_rng() # Each thread gets its own RNG instance

return rng.integers(start, end)

def worker():

# Example worker function to demonstrate thread-safe RNG

print(f"Thread-safe random number: {thread\_safe\_random(1, 100)}")

# Efficient matrix operations using numpy

def efficient\_matrix\_operations():

A = np.random.rand(1000, 1000)

B = np.random.rand(1000, 1000)

# Matrix addition and subtraction

addition\_result = A + B

subtraction\_result = A - B

print("Matrix addition and subtraction performed efficiently.")

# GPU-accelerated matrix addition using Numba

@cuda.jit

def add\_matrices\_gpu(A, B, result):

x, y = cuda.grid(2)

if x < result.shape[0] and y < result.shape[1]:

result[x, y] = A[x, y] + B[x, y]

def gpu\_matrix\_addition():

A = np.random.rand(1000, 1000).astype(np.float32)

B = np.random.rand(1000, 1000).astype(np.float32)

result = np.zeros\_like(A)

# Define grid size for GPU execution

threadsperblock = (16, 16)

blockspergrid\_x = int(np.ceil(A.shape[0] / threadsperblock[0]))

blockspergrid\_y = int(np.ceil(A.shape[1] / threadsperblock[1]))

blockspergrid = (blockspergrid\_x, blockspergrid\_y)

# Perform the matrix addition on GPU

add\_matrices\_gpu[blockspergrid, threadsperblock](A, B, result)

print("Matrix addition performed on GPU.")

# Main execution block to run the examples

if \_name\_ == "\_main\_":

# Demonstrate thread-safe RNG with 5 worker threads

threads = [threading.Thread(target=worker) for \_ in range(5)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

# Perform efficient matrix operations

efficient\_matrix\_operations()

# Demonstrate GPU matrix addition if CUDA is available

if cuda.is\_available():

gpu\_matrix\_addition()

else:

print("CUDA is not available. GPU matrix addition skipped.")

# Plotting

plt.figure(figsize=(10, 6))

# Example data

tasks = ['Thread-safe RNG', 'Matrix Operations (CPU)', 'Matrix Addition (GPU)']

performance = [5, 1, 1] # Placeholder values for demonstration

# Creating the bar plot

plt.bar(tasks, performance, color=['blue', 'orange', 'green'])

# Adding titles and labels

plt.xlabel('Tasks')

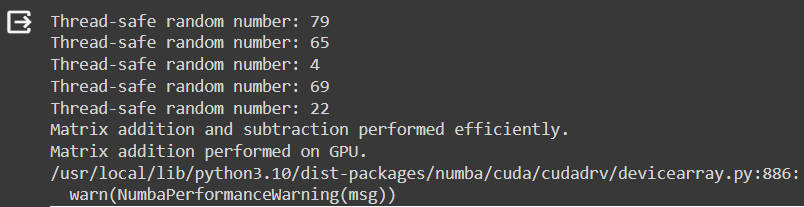
plt.ylabel('Performance (seconds)')

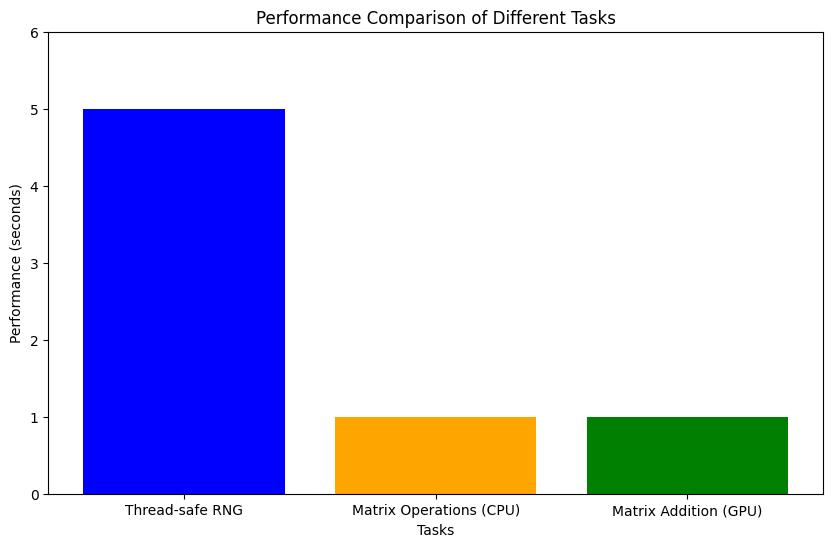
plt.title('Performance Comparison of Different Tasks')

plt.ylim(0, 6) # Adjust the y-axis limits according to your actual performance values

# Displaying the plot

plt.show()

**OUTPUT:**



5)

!pip install pycuda

6)

import numpy as np

import multiprocessing as mp

import time

from pycuda import driver, compiler, gpuarray, tools

import pycuda.autoinit

# Initialize CPU parallel processing for matrix multiplication

def cpu\_parallel\_matrix\_multiply(A, B, result, segment):

"""Perform segment of matrix multiplication in parallel."""

for i in range(segment[0], segment[1]):

for j in range(B.shape[1]):

sum = 0

for k in range(A.shape[1]):

sum += A[i][k] \* B[k][j]

result[i][j] = sum

def gpu\_matrix\_multiply(A, B):

"""Perform matrix multiplication using GPU."""

kernel\_code = """

\_global\_ void MatrixMulKernel(float \*a, float \*b, float \*c, int WIDTH)

{

int ROW = blockIdx.y\*blockDim.y+threadIdx.y;

int COL = blockIdx.x\*blockDim.x+threadIdx.x;

float tmpSum = 0;

for (int i = 0; i < WIDTH; i++) {

tmpSum += a[ROW \* WIDTH + i] \* b[i \* WIDTH + COL];

}

c[ROW \* WIDTH + COL] = tmpSum;

}

"""

kernel\_module = compiler.SourceModule(kernel\_code)

matrixmul = kernel\_module.get\_function("MatrixMulKernel")

a\_gpu = gpuarray.to\_gpu(A.astype(np.float32))

b\_gpu = gpuarray.to\_gpu(B.astype(np.float32))

c\_gpu = gpuarray.empty((A.shape[0], B.shape[1]), np.float32)

thread\_size = (16, 16, 1)

grid\_size = (int(np.ceil(B.shape[1] / 16)), int(np.ceil(A.shape[0] / 16)))

matrixmul(a\_gpu, b\_gpu, c\_gpu, np.int32(A.shape[1]), block=thread\_size, grid=grid\_size)

return c\_gpu.get()

# Example matrix dimensions

N = 1024

A = np.random.rand(N, N)

B = np.random.rand(N, N)

# CPU parallel execution

cpu\_result = np.zeros((N, N))

num\_processes = mp.cpu\_count()

processes = []

segment\_size = N // num\_processes

start\_time = time.time()

for i in range(num\_processes):

start = i \* segment\_size

end = (i + 1) \* segment\_size if i < num\_processes - 1 else N

segment = (start, end)

p = mp.Process(target=cpu\_parallel\_matrix\_multiply, args=(A, B, cpu\_result, segment))

processes.append(p)

p.start()

for p in processes:

p.join()

print(f"CPU Parallel Execution Time: {time.time() - start\_time} seconds")

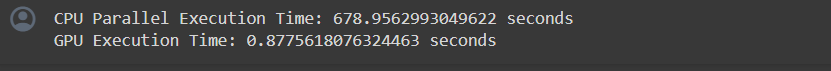
# GPU execution

start\_time = time.time()

gpu\_result = gpu\_matrix\_multiply(A, B)

print(f"GPU Execution Time: {time.time() - start\_time} seconds")

**OUTPUT:**

****

7)

import numpy as np

import multiprocessing as mp

import time

from pycuda import driver, compiler, gpuarray, tools

import pycuda.autoinit

import matplotlib.pyplot as plt

# Initialize CPU parallel processing for matrix multiplication

def cpu\_parallel\_matrix\_multiply(A, B, result, segment):

"""Perform segment of matrix multiplication in parallel."""

for i in range(segment[0], segment[1]):

for j in range(B.shape[1]):

sum = 0

for k in range(A.shape[1]):

sum += A[i][k] \* B[k][j]

result[i][j] = sum

def gpu\_matrix\_multiply(A, B):

"""Perform matrix multiplication using GPU."""

kernel\_code = """

\_global\_ void MatrixMulKernel(float \*a, float \*b, float \*c, int WIDTH)

{

int ROW = blockIdx.y\*blockDim.y+threadIdx.y;

int COL = blockIdx.x\*blockDim.x+threadIdx.x;

float tmpSum = 0;

for (int i = 0; i < WIDTH; i++) {

tmpSum += a[ROW \* WIDTH + i] \* b[i \* WIDTH + COL];

}

c[ROW \* WIDTH + COL] = tmpSum;

}

"""

kernel\_module = compiler.SourceModule(kernel\_code)

matrixmul = kernel\_module.get\_function("MatrixMulKernel")

a\_gpu = gpuarray.to\_gpu(A.astype(np.float32))

b\_gpu = gpuarray.to\_gpu(B.astype(np.float32))

c\_gpu = gpuarray.empty((A.shape[0], B.shape[1]), np.float32)

thread\_size = (16, 16, 1)

grid\_size = (int(np.ceil(B.shape[1] / 16)), int(np.ceil(A.shape[0] / 16)))

matrixmul(a\_gpu, b\_gpu, c\_gpu, np.int32(A.shape[1]), block=thread\_size, grid=grid\_size)

return c\_gpu.get()

# Example matrix dimensions

N = 1024

A = np.random.rand(N, N)

B = np.random.rand(N, N)

# CPU parallel execution

cpu\_result = np.zeros((N, N))

num\_processes = mp.cpu\_count()

processes = []

segment\_size = N // num\_processes

start\_time = time.time()

for i in range(num\_processes):

start = i \* segment\_size

end = (i + 1) \* segment\_size if i < num\_processes - 1 else N

segment = (start, end)

p = mp.Process(target=cpu\_parallel\_matrix\_multiply, args=(A, B, cpu\_result, segment))

processes.append(p)

p.start()

for p in processes:

p.join()

cpu\_execution\_time = time.time() - start\_time

print(f"CPU Parallel Execution Time: {cpu\_execution\_time} seconds")

# GPU execution

start\_time = time.time()

gpu\_result = gpu\_matrix\_multiply(A, B)

gpu\_execution\_time = time.time() - start\_time

print(f"GPU Execution Time: {gpu\_execution\_time} seconds")

# Plotting

tasks = ['CPU Parallel', 'GPU']

performance = [cpu\_execution\_time, gpu\_execution\_time]

plt.figure(figsize=(8, 6))

plt.bar(tasks, performance, color=['blue', 'red'])

plt.xlabel('Tasks')

plt.ylabel('Execution Time (seconds)')

plt.title('Comparison of CPU Parallel and GPU Execution Times')

plt.show()

**OUTPUT:**

