```
%%writefile evaluate_fitness_batch.py
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load dataset
df = pd.read_csv("Crop_recommendation.csv")
X = df.drop(columns="label")  # Ensure all features are included
y = df["label"]

# Read population (batch of chromosomes) from a file
population = np.loadtxt("population_batch.txt", delimiter=",", dtype=int)
print(f"Population Shape: {population.shape}")

# Validate chromosome length
if population.shape[1] != X.shape[1]:
    raise ValueError(f"Chromosome length {population.shape[1]} does not match the number of

# Initialize list to store fitness values
fitness_results = []

# Loop through each chromosome in the population
for chromosome_id, chromosome in enumerate(population, start=1):
    print(f"Chromosome {chromosome_id}: {chromosome}")

    # Get selected features
    selected_features = np.where(chromosome == 1)[0]
    print("Selected Features:", selected_features)

    # Handle case where no features are selected
    if len(selected_features) == 0:
        print(f"Chromosome {chromosome_id}: No features selected. Assigning fitness 0.")
        fitness_results.append(0.0)
        continue

    # Select features based on chromosome
    X_selected = X.iloc[:, selected_features]

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.3, random

    # Train the model
    rf = RandomForestClassifier(n_estimators=50, random_state=42, criterion="entropy")
    rf.fit(X_train, y_train)

    # Predict and calculate accuracy
    y_pred = rf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Chromosome {chromosome_id} Accuracy:", accuracy)

    # Store fitness result
    fitness_results.append(accuracy)

# Save all fitness results to a file
np.savetxt("fitness_results_batch.txt", fitness_results, fmt="%.4f")
print("Fitness results saved.")
```

```
Overwriting evaluate_fitness_batch.py
```

```
%%writefile cudamain.cu

#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

#define POP_SIZE 200
#define CHROMOSOME_LENGTH 7
#define NUM_GENERATIONS 50
#define MUTATION_RATE 0.01f
#define MAX_STAGNATION 3
#define BLOCK_SIZE 256

typedef struct {
    int id;
    int genes[CHROMOSOME_LENGTH];
    double fitness;
} Individual;

// CUDA kernel for random chromosome generation
__global__ void generate_population_kernel(int* population, curandState* states,
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < pop_size) {
        curandState localState = states[idx];
        for (int i = 0; i < chromosome_length; i++) {
            population[idx * chromosome_length + i] = curand(&localState) % 2;
        }
        states[idx] = localState;
    }
}

// CUDA kernel for mutation
__global__ void mutation_kernel(int* population, curandState* states, float mutat
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < pop_size * chromosome_length) {
        int chr_idx = idx / chromosome_length;
        int gene_idx = idx % chromosome_length;
        curandState localState = states[chr_idx];
        float r = curand_uniform(&localState);
        if (r < mutation_rate) {
            population[idx] = 1 - population[idx];
        }
        states[chr_idx] = localState;
    }
}

// CUDA kernel for crossover
__global__ void crossover_kernel(int* population, int* new_population, int pop_si
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < pop_size / 2) {
        int parent1_idx = 2 * idx;
        int parent2_idx = 2 * idx + 1;

        int swap_start = chromosome_length - 3;

        // Pointers to parents and offspring
        int* parent1 = &population[parent1_idx * chromosome_length];
        int* parent2 = &population[parent2_idx * chromosome_length];
        int* offspring1 = &new_population[parent1_idx * chromosome_length];
        int* offspring2 = &new_population[parent2_idx * chromosome_length];

        // Perform three-point crossover
        for (int i = 0; i < chromosome_length; i++) {
            if (i < swap_start) {
                offspring1[i] = parent1[i];
                offspring2[i] = parent2[i];
            } else {
                offspring1[i] = parent2[i];
                offspring2[i] = parent1[i];
            }
        }
    }
}

// Function to initialize random states
__global__ void init_random_states(curandState* states, unsigned long seed) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < POP_SIZE) {
        curand_init(seed, idx, 0, &states[idx]);
    }
}

int main() {
    // Allocate device memory
    int *d_population, *d_new_population;
    curandState *d_states;
    cudaMalloc(&d_population, POP_SIZE * CHROMOSOME_LENGTH * sizeof(int));
    cudaMalloc(&d_new_population, POP_SIZE * CHROMOSOME_LENGTH * sizeof(int));
    cudaMalloc(&d_states, POP_SIZE * sizeof(curandState));

    // Initialize random states
    int blocks = (POP_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
    init_random_states<<<blocks, BLOCK_SIZE>>>(d_states, time(NULL));
    cudaDeviceSynchronize();

    // Generate initial population
    generate_population_kernel<<<blocks, BLOCK_SIZE>>>(d_population, d_states, PO
    cudaDeviceSynchronize();

    // Allocate host memory
    Individual* population = (Individual*)malloc(POP_SIZE * sizeof(Individual));
    Individual* new_population = (Individual*)malloc(POP_SIZE * sizeof(Individual
    int* h_population = (int*)malloc(POP_SIZE * CHROMOSOME_LENGTH * sizeof(int));
    cudaMemcpy(h_population, d_population, POP_SIZE * CHROMOSOME_LENGTH * sizeof(

    // Initialize population structs
    for (int i = 0; i < POP_SIZE; i++) {
        population[i].id = i + 1;
        memcpy(population[i].genes, &h_population[i * CHROMOSOME_LENGTH], CHROMOS
    }

    double best_fitness = -1.0;
    int stagnation_count = 0;

    for (int generation = 0; generation < NUM_GENERATIONS; generation++) {
        printf("\n=== Generation %d ===\n", generation);

        // Write population to file
        FILE* file = fopen("population_batch.txt", "w");
        if (!file) {
            perror("Error opening batch file");
            exit(EXIT_FAILURE);
        }
        for (int i = 0; i < POP_SIZE; i++) {
            for (int j = 0; j < CHROMOSOME_LENGTH; j++) {
                fprintf(file, "%d", population[i].genes[j]);
                if (j < CHROMOSOME_LENGTH - 1) fprintf(file, ",");
            }
            fprintf(file, "\n");
        }
        fclose(file);

        // Execute Python script
        system("python3 evaluate_fitness_batch.py");

        // Read fitness results
        file = fopen("fitness_results_batch.txt", "r");
        if (!file) {
            perror("Error opening fitness results batch file");
            exit(EXIT_FAILURE);
        }
        for (int i = 0; i < POP_SIZE; i++) {
            fscanf(file, "%lf", &population[i].fitness);
        }
        fclose(file);

        // Find the best individual
        double current_best_fitness = -1.0;
        int best_idx = -1;
        for (int i = 0; i < POP_SIZE; i++) {
            if (population[i].fitness > current_best_fitness) {
                current_best_fitness = population[i].fitness;
                best_idx = i;
            }
        }

        printf("Best Fitness: %.4f\n", current_best_fitness);

        // Check for stagnation
         if (current_best_fitness == best_fitness) {
        stagnation_count++;
        printf("No improvement in generation %d. Stagnation count: %d\n", generat
        if (stagnation_count >= MAX_STAGNATION) {
            printf("Stopping early due to stagnation: No improvement for %d gener
            break;
        }
    } else {
        best_fitness = current_best_fitness;
        stagnation_count = 0;
    }

        // Preserve elite
        new_population[0] = population[best_idx];

        // Perform crossover and mutation
        cudaMemcpy(d_population, h_population, POP_SIZE * CHROMOSOME_LENGTH * siz
        crossover_kernel<<<blocks, BLOCK_SIZE>>>(d_population, d_new_population,
        cudaDeviceSynchronize();
        mutation_kernel<<<blocks, BLOCK_SIZE>>>(d_new_population + CHROMOSOME_LEN
        cudaDeviceSynchronize();

        // Copy results back
        cudaMemcpy(h_population, d_new_population, POP_SIZE * CHROMOSOME_LENGTH *

        for (int i = 1; i < POP_SIZE; i++) {
            memcpy(population[i].genes, &h_population[i * CHROMOSOME_LENGTH], CHR
        }
        population[0] = new_population[0];
    }

    // Cleanup
    cudaFree(d_population);
    cudaFree(d_new_population);
    cudaFree(d_states);
    free(h_population);
    free(population);
    free(new_population);

    return 0;
}
```

```
Overwriting cudamain.cu
```

```
!nvcc -o cudamain cudamain.cu
```

```
cudamain.cu(37): warning #177-D: variable "gene_idx" was declared but never referenced
          int gene_idx = idx % chromosome_length;
              ^

Remark: The warnings can be suppressed with "-diag-suppress <warning-number>"
```

```
import time

t = time.time()
!./cudamain
```

```
Selected Features: [0 1 2 4 6]
Chromosome 174 Accuracy: 0.9878787878787879
Chromosome 175: [0 0 1 0 0 1 0]
Selected Features: [2 5]
Chromosome 175 Accuracy: 0.4681818181818182
Chromosome 176: [1 1 1 0 1 1 0]
Selected Features: [0 1 2 4 5]
Chromosome 176 Accuracy: 0.9409090909090909
Chromosome 177: [1 0 1 1 0 1 1]
Selected Features: [0 2 3 5 6]
Chromosome 177 Accuracy: 0.9575757575757575
Chromosome 178: [1 0 1 1 1 0 0]
Selected Features: [0 2 3 4]
Chromosome 178 Accuracy: 0.9166666666666666
Chromosome 179: [1 0 1 0 0 0 1]
Selected Features: [0 2 6]
Chromosome 179 Accuracy: 0.8424242424242424
Chromosome 180: [0 0 1 1 1 1 1]
Selected Features: [2 3 4 5 6]
Chromosome 180 Accuracy: 0.9742424242424242
Chromosome 181: [1 1 1 1 1 0 0]
Selected Features: [0 1 2 3 4]
Chromosome 181 Accuracy: 0.956060606060606
Chromosome 182: [1 1 1 0 1 0 0]
Selected Features: [0 1 2 4]
Chromosome 182 Accuracy: 0.9363636363636364
Chromosome 183: [0 1 0 0 1 0 0]
Selected Features: [1 4]
Chromosome 183 Accuracy: 0.6484848484848484
Chromosome 184: [0 1 0 1 0 0 1]
Selected Features: [1 3 6]
Chromosome 184 Accuracy: 0.8545454545454545
```

```
delta_t = time.time()-t
print("Time taken for total execution of parallel code:", delta_t)
parallel_time = delta_t
```

```
Time taken for total execution of parallel code: 195.64607548713684
```

```
!nvprof --print-summary ./cudamain
```

```
Selected Features: [0 1 2 4]
Chromosome 194 Accuracy: 0.9363636363636364
Chromosome 195: [0 0 0 0 1 1 1]
Selected Features: [4 5 6]
Chromosome 195 Accuracy: 0.8621212121212121
Chromosome 196: [0 0 1 1 1 0 1]
Selected Features: [2 3 4 6]
Chromosome 196 Accuracy: 0.9742424242424242
Chromosome 197: [0 0 0 1 0 1 1]
Selected Features: [3 5 6]
Chromosome 197 Accuracy: 0.75
Chromosome 198: [1 1 0 1 0 0 0]
Selected Features: [0 1 3]
Chromosome 198 Accuracy: 0.7090909090909091
Chromosome 199: [1 0 1 1 0 1 1]
Selected Features: [0 2 3 5 6]
Chromosome 199 Accuracy: 0.9575757575757575
Chromosome 200: [1 0 0 0 0 1 0]
Selected Features: [0 5]
Chromosome 200 Accuracy: 0.296969696969697
Fitness results saved.
Best Fitness: 0.9924
No improvement in generation 3. Stagnation count: 3
Stopping early due to stagnation: No improvement for 3 generations.
==19859== Profiling application: ./cudamain
==19859== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   80.25%  200.26us         1  200.26us  200.26us  200.26us  init_ran
                    7.86%  19.616us         3  6.5380us  6.4320us  6.7200us  crossove
                    4.07%  10.144us         3  3.3810us  3.3280us  3.4560us  mutation
                    3.44%  8.5760us         4  2.1440us  1.9840us  2.3680us  [CUDA me
                    3.04%  7.5840us         1  7.5840us  7.5840us  7.5840us  generate
                    1.35%  3.3600us         3  1.1200us  1.1200us  1.1200us  [CUDA me
      API calls:   96.63%  176.87ms         3  58.956ms  3.1930us  176.86ms  cudaMall
                    2.74%  5.0171ms         8  627.13us  8.6540us  4.8334ms  cudaLaun
                    0.18%  329.04us         3  109.68us  8.6130us  225.35us  cudaFree
                    0.18%  328.66us         7  46.950us  24.331us  77.251us  cudaMemc
                    0.15%  282.94us         8  35.367us  6.4690us  219.88us  cudaDevi
                    0.09%  158.86us       114  1.3930us     170ns  58.221us  cuDevice
                    0.01%  26.368us         1  26.368us  26.368us  26.368us  cuDevice
                    0.01%  12.713us         1  12.713us  12.713us  12.713us  cuDevice
                    0.00%  5.5500us         1  5.5500us  5.5500us  5.5500us  cuDevice
                    0.00%  1.6420us         3     547ns     180ns  1.1050us  cuDevice
                    0.00%     967ns         2     483ns     215ns     752ns  cuDevice
                    0.00%     654ns         1     654ns     654ns     654ns  cuModule
                    0.00%     265ns         1     265ns     265ns     265ns  cuDevice
```