

*School of Arts and Sciences*

*Department of Computer Science and Mathematics*

**Optimizing Feature Selection Using Genetic Algorithms**

**Accelerated with CUDA**

Abas Ismail **–** 201901521

Ikram Bichbich **–** 202208655

Omar Shatila **–** 202000600

*CSC 447:* Parallel Programming for multicore and cluster systems

Dr. Rachad Atat

**December 12h, 2024**

Table of Contents

[I. What is Genetic Algorithm: 1](#_Toc185001957)

[II. How does it work: 1](#_Toc185001958)

[**1.** **Selection:** 1](#_Toc185001959)

[**2.** **Crossover:** 1](#_Toc185001960)

[**3.** **Mutation:** 2](#_Toc185001961)

[III. Sequential Code: 2](#_Toc185001962)

[ Constants: 2](#_Toc185001963)

[ Helper Functions: 3](#_Toc185001964)

[IV. Parallelization using CUDA C: 4](#_Toc185001965)

[A. Pseudocode: 4](#_Toc185001966)

[B. Parallelization Techniques: 6](#_Toc185001967)

[**1.** **Data Distribution: Domain Decomposition:** 6](#_Toc185001968)

[**2.** **Collective Communication and Reduction:** 6](#_Toc185001969)

[**3.** **Synchronization:** 6](#_Toc185001970)

[**4.** **Load Balancing:** 7](#_Toc185001971)

[**5.** **Specific CUDA Optimizations:** 7](#_Toc185001972)

[C. Numerical Analysis: 7](#_Toc185001973)

[V. Conclusion: 8](#_Toc185001974)

[ Challenges and Limitations: 8](#_Toc185001975)

[ Future Work: 9](#_Toc185001976)

[VI. Project Management 9](#_Toc185001977)

[ Team Roles and Contributions: 9](#_Toc185001978)

[ **Abbas, Ikram, and Omar** 9](#_Toc185001979)

[ **Ikram:** 9](#_Toc185001980)

[ **Omar:** 9](#_Toc185001981)

[ **Abbas:** 9](#_Toc185001982)

[ Communication Methods: 9](#_Toc185001983)

[ Project Timeline: 9](#_Toc185001984)

[ **Week 1:** 9](#_Toc185001985)

[ **Week 2:** 9](#_Toc185001986)

[ **Week 3:** 10](#_Toc185001987)

[ **Week 4:** 10](#_Toc185001988)

[ **Week 5:** 10](#_Toc185001989)

[VII. Link for The Code: 10](#_Toc185001990)

[VIII. References: 11](#_Toc185001991)

**Abstract**

Feature selection plays a critical role in machine learning as it directly impacts model performance and complexity. Selecting the optimal subset of features is a challenging task due to the dependencies that often exist between features. To address this problem, we applied a **Genetic Algorithm (GA)**, an evolutionary optimization technique, to identify the most relevant features for the model. The Genetic Algorithm explores multiple subsets of features represented as chromosomes, optimizing the selection process iteratively.

Initially, we implemented the GA in a **serial code**, which demonstrated good accuracy but suffered from significant time consumption, especially with a population size of 50. To improve computational efficiency, we parallelized the Genetic Algorithm using **CUDA** to leverage GPU acceleration. This parallel implementation enabled us to scale the population to a very large populations, significantly reducing execution time while maintaining high accuracy.

Both the serial and parallel implementations were tested, and the final optimized chromosome achieved an accuracy of **0.92**, indicating excellent model performance with the selected features. The key difference, however, was in the execution time: the **parallel code** outperformed the serial version by a considerable margin, enabling faster processing and scalability for larger populations.

The results demonstrate that the parallelized Genetic Algorithm not only maintained model performance but also **reduced execution time dramatically**, showcasing the advantage of CUDA acceleration for large-scale optimization tasks in feature selection.

**1-Introduction:**

**Problem Definition**

Feature selection is a critical task in machine learning, where the goal is to identify the most relevant features from a dataset to improve model performance while reducing computational complexity. Selecting an optimal subset of features is challenging because of potential **dependencies and redundancies** among features, making it difficult to determine which features contribute most to the model's accuracy.

In this project, we addressed the feature selection problem for a **Random Forest-based recommendation model**. The model aims to predict the best recommendations for a given task, and selecting the correct features is essential to ensure high accuracy while minimizing the number of inputs.

**Dataset Details:**  We utilized the **Crop Recommendation Dataset** from Kaggle, which includes:

* **Source**: Kaggle (Crop Recommendation Dataset)
* **Size**: 2,200 entries
* **Features**:
* **N (Nitrogen)**: Amount of nitrogen in the soil, a key nutrient for leaf growth.
* **P (Phosphorus)**: Amount of phosphorus in the soil, critical for root and flower development.
* **K (Potassium)**: Potassium level in the soil, essential for plant health and resilience.
* **Temperature**: Average temperature in the region, affecting plant growth cycles.
* **Humidity**: Humidity levels, influencing water loss and transpiration rates.
* **pH**: Soil pH level, determining nutrient availability and suitability for different crops.
* **Rainfall**: Annual or seasonal rainfall, indicating water availability.
* **Label**: The recommended crop type based on the above features.

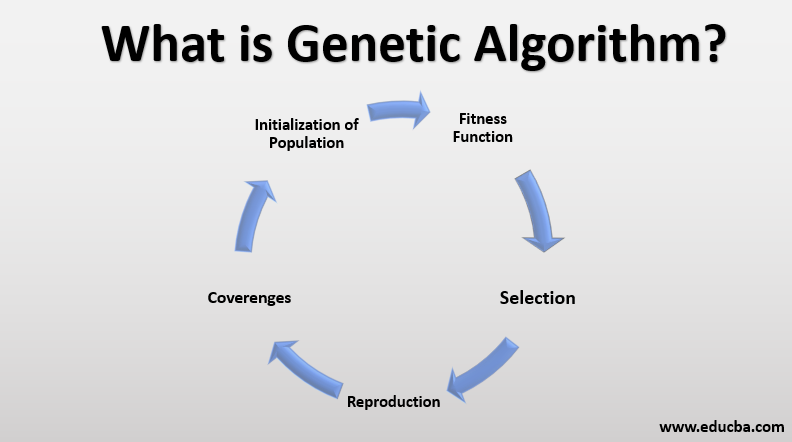
You can access this dataset through the link for further details: <https://www.kaggle.com/datasets/atharvaingle/crop-recommendation-dataset>

However, manually selecting features is impractical due to the large search space and complex interdependencies between features.

To solve this, we implemented a **Genetic Algorithm (GA)**, a heuristic optimization technique inspired by natural evolution. GA explores various combinations of features (represented as chromosomes) and evaluates their effectiveness iteratively to identify the optimal subset. The focus was on optimizing the model's accuracy while maintaining computational efficiency.

**What is Genetic Algorithm:**

This project explores the concept of the Genetic Algorithm (GA), an optimization technique inspired by the principles of natural selection and biological evolution. GAs iteratively improve a population of candidate solutions through processes like selection, crossover, and mutation, making them effective for solving complex problems with nonlinear, stochastic, or discontinuous objective functions. The project also explores how parallel computing platforms like CUDA, which harness the power of GPUs, can significantly speed up GA performance, especially for large-scale datasets. By utilizing CUDA, the project aims to demonstrate the effectiveness of genetic algorithms in solving computationally intensive optimization problems while optimizing performance through parallelization.



# **How does it work:**

The genetic algorithm begins with an initial population of individuals, typically generated randomly. Each individual represents a potential solution to the optimization problem, encoded as a set of genes (e.g., a binary string).

* The algorithm then progresses through a series of iterations called **generations**, applying the following key processes:

1. **Selection:** Individuals with higher fitness values are chosen as parents. Fitness is calculated based on how well an individual solves the optimization problem. This ensures that favorable traits are passed to the next generation.
2. **Crossover:** Selected parents undergo crossover (recombination) to create offspring. This involves swapping segments of genes between two parents to produce new individuals, promoting diversity and innovation in the population.
3. **Mutation:** To maintain genetic diversity, random mutations are introduced in the offspring's genes with a small probability. This prevents premature convergence and helps explore the solution space more effectively.

These processes mimic natural evolution, where individuals compete for survival, reproduce, and adapt over time. Over successive generations, the population "evolves" toward an optimal solution. The algorithm halts when it meets predefined criteria, such as achieving a target fitness or reaching a maximum number of generations.

### **CUDA Overview**

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by **NVIDIA** that enables significant performance improvements by leveraging the power of Graphics Processing Units (GPUs). Unlike CPUs, which process tasks sequentially, GPUs are designed to execute thousands of threads concurrently, making them highly efficient for tasks that require extensive computations.

In this project, we used CUDA to **parallelize the Genetic Algorithm (GA)** for feature selection. The motivation behind this approach stems from the limitations of the serial implementation, which suffers from high execution time when dealing with larger populations and complex datasets. CUDA allows us to exploit GPU parallelism to:

* Process multiple chromosomes in the population simultaneously.
* Accelerate the computation of fitness evaluations, mutations, and crossovers.
* Achieve faster convergence while maintaining solution quality.

By parallelizing key components of the GA, such as **population generation** and **fitness evaluation**, we achieved substantial reductions in computation time compared to the serial implementation, enabling the use of larger population sizes and more iterations.

### **Project Objectives**

The primary goals of this project are as follows:

1. **Optimization of Feature Selection**: To identify the most relevant subset of features that improves the performance of a **Random Forest-based recommendation model** while reducing computational complexity.
2. **Implementation of Genetic Algorithm (GA)**: To apply the Genetic Algorithm as a heuristic optimization technique for feature selection, using population-based search and evolutionary operations (selection, mutation, crossover) to identify the optimal feature set.
3. **CUDA Parallelization for Performance Enhancement**: To parallelize the Genetic Algorithm using **CUDA** to improve execution time and scalability. Also, Leverage GPU capabilities to handle larger populations and iterations efficiently.
4. **Performance Comparison**: Compare the execution time, solution quality, and convergence speed between the **serial implementation** and the **CUDA-accelerated implementation**. In addition, Demonstrate how CUDA acceleration reduces computational costs while maintaining model accuracy.
5. **Achieve High Model Accuracy**:

Ensure that the final selected features lead to a model accuracy of at least **0.92**, balancing performance and computational efficiency.

#### **3-Algorithm Implementation:**

**Serial Implementation**:

## **Constants:**

* + **POP\_SIZE:** Defines the size of the population (number of individuals in each generation).
* **Value:** 50
* **CHROMOSOME\_LENGTH:** Specifies the length of each individual's chromosome (number of genes).
  + **Value:** 7
* **NUM\_GENERATIONS:** Sets the number of generations for the genetic algorithm to run.
  + **Value:** 50
* **MUTATION\_RATE:** Determines the probability of a mutation occurring for each gene in a chromosome.
  + **Value:** 0.01
* **MAX\_STAGNATION:** Sets the maximum number of generations allowed without improvement in fitness before stopping early due to stagnation.
  + **Value:** 3

## **Helper Functions:**

* **generate\_chromosome(int chromosome[], int length):** Generates a random chromosome by assigning binary values (0 or 1) to each gene.
* **evaluate\_fitness(int chromosome[], int length, int chromosome\_id):** Calculates the fitness of a chromosome by writing it to a file, executing a Python script, and reading the fitness value from the output file.
* **roulette\_wheel\_selection(Individual population[], int population\_size):** Implements roulette wheel selection to choose a parent for crossover based on fitness proportion.
* **three\_point\_crossover(int parent1[], int parent2[], int offspring1[], int offspring2[], int length):** Performs a three-point crossover to produce offspring by swapping the last three genes between two parents.
* **mutate(int chromosome[], int length, double mutation\_rate)**: Introduces mutations to a chromosome by flipping bits with a probability equal to the mutation rate.

### **Structure and Components**

**1-Feature Selection Task**: The goal is to find the best subset of features for a Random Forest model using **Genetic Algorithm (GA)**. Each subset of features is represented as a **chromosome**.

**2-Steps in the Serial Implementation**:

**Population Initialization**: A random population of chromosomes (binary vectors) is generated, where each gene represents whether a feature is selected (1) or not (0).

**Fitness Evaluation**: Each chromosome's fitness is evaluated using a **Python script** (evaluate\_fitness.py), which trains and tests a Random Forest model. The fitness score is the model’s accuracy.

**Selection**: Parents are selected using **Roulette Wheel Selection** based on fitness values.

**Crossover**: Offspring are generated using **Three-Point Crossover** to combine features of parent chromosomes.

**Mutation**: Random genes are flipped with a small mutation probability to introduce diversity.

**Termination**: The algorithm stops after a set number of generations or if there is no improvement for a specified number of generations (stagnation).

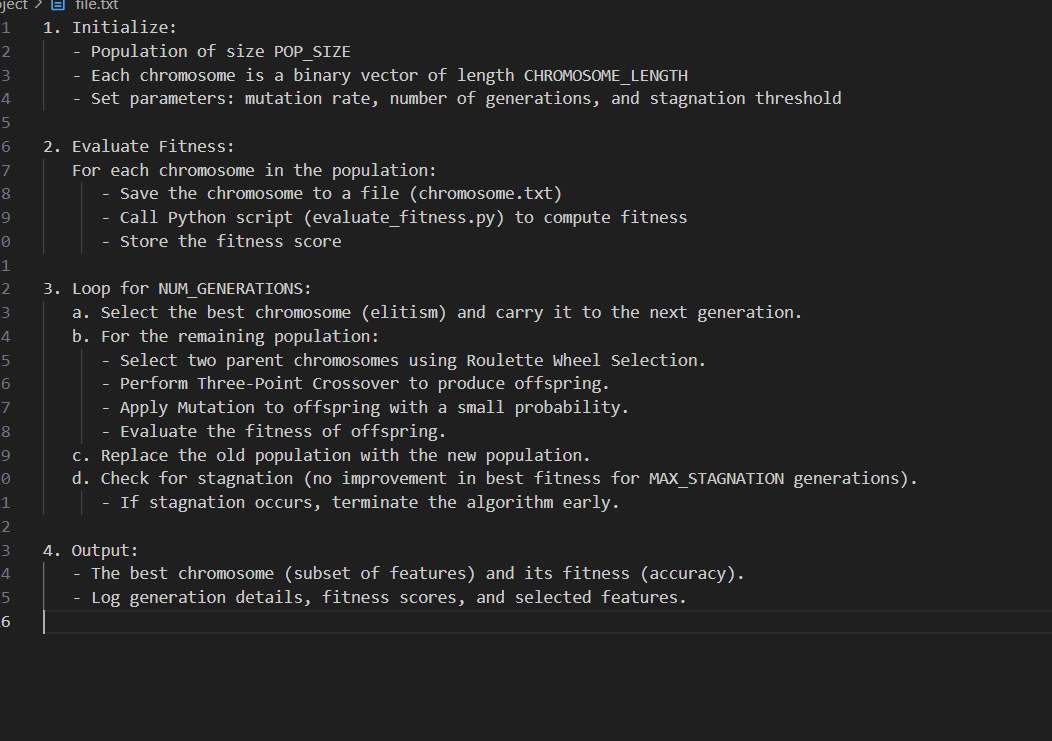
**3-Files Used**:

**evaluate\_fitness.py**: Python script to evaluate the accuracy of a chromosome using Random Forest.

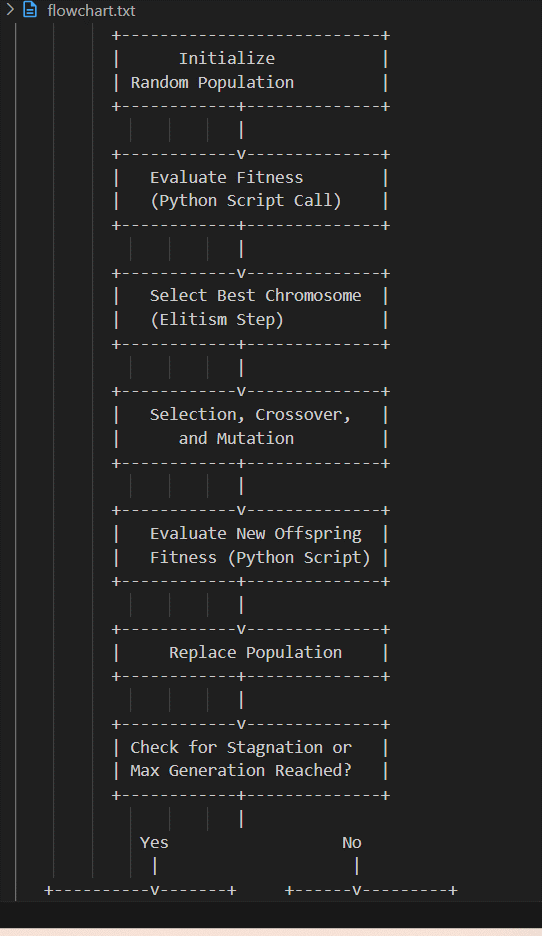
**main.c**: The main C program implementing the GA logic, including selection, crossover, mutation, and fitness evaluation.

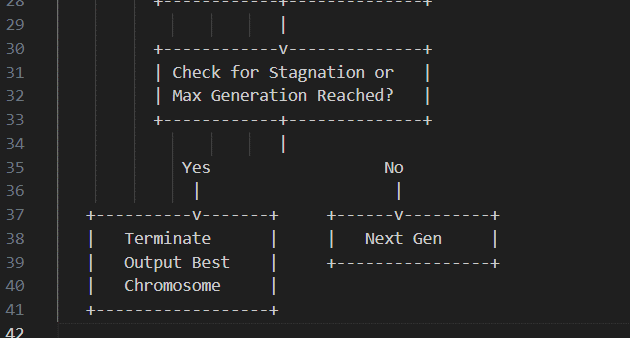
## **Pseudocode:**

Main Algorithm (Genetic Algorithm)



**Flowchart:**





### **Explanation of Key Components in the Code**

**Fitness Evaluation**: Chromosomes are evaluated by saving them to chromosome.txt and calling evaluate\_fitness.py to compute model accuracy using a Random Forest Classifier.

**Selection**: **Roulette Wheel Selection** is used to choose parent chromosomes probabilistically based on fitness.

**Crossover**: A **Three-Point Crossover** is applied to combine genetic material from two parents.

**Mutation**: Random genes are flipped (0 ↔ 1) with a probability defined by MUTATION\_RATE.

**Elitism**: The best chromosome (highest fitness) is carried over to the next generation.

#### **4-CUDA Acceleration:**

## **Parallelization Strategy:**

The Genetic Algorithm (GA) was parallelized using **CUDA** to accelerate computationally expensive tasks that arise in the feature selection optimization process. Given the independent nature of GA operations like population initialization, mutation, crossover, and fitness evaluation, these tasks were efficiently distributed across GPU threads to leverage CUDA’s parallel computing capabilities. Below is a detailed breakdown of the parallelized components:

### **1-Population Initialization**

**Kernel**: generate\_population\_kernel

**What It Does**:

* This kernel generates an initial population of chromosomes, where each chromosome is represented as a binary vector (0s and 1s).
* Each gene in the chromosome is assigned a random value (0 or 1) using the **curand** random number generator.

**Why Parallelize It?**

* Population initialization is inherently parallel, as each gene in a chromosome can be generated independently.
* By mapping genes to GPU threads, the entire population is initialized concurrently, drastically reducing execution time.

**Thread Mapping**:

* Each **thread** is responsible for generating a single gene within a chromosome.
* Grid and block sizes are chosen dynamically based on the population size and chromosome length.

### **2-Mutation Operation**

**Kernel**: mutation\_kernel

**What It Does**:

* The kernel applies **mutation** to individual genes in each chromosome based on a given mutation probability.
* If the mutation condition is met (using curand), the gene value is flipped (0 → 1 or 1 → 0).

**Why Parallelize It?**

* Mutation is applied independently to each gene in the population, making it perfectly suited for parallel execution.
* GPU threads ensure that the mutation process scales efficiently with larger populations.

**Thread Mapping**:

* Each **thread** corresponds to a single gene within the chromosome.

### **3-Crossover Operation**

**Kernel**: crossover\_kernel

**What It Does**:

* The kernel performs **three-point crossover** between pairs of parent chromosomes.
* The crossover operation splits the parents into segments at three points and swaps segments to produce offspring.

**Why Parallelize It?**

* Each chromosome pair can be processed independently, enabling parallel computation of offspring.
* This reduces the time required to generate the next population significantly.

**Thread Mapping**:

* Each **thread** is assigned to a pair of parent chromosomes, allowing multiple crossovers to occur simultaneously.

### **4-Fitness Evaluation (Batch Processing)**

**What It Does**:

* Fitness evaluation, which involves measuring the performance of a chromosome on the machine learning model, is handled in **batches** to optimize data transfer between the CPU and GPU.
* Chromosomes are transferred to the host (CPU) in **coalesced memory blocks** for fitness evaluation, minimizing memory latency and overhead.

**Why Parallelize It?**

* Although fitness evaluation itself is not GPU-accelerated (handled by the Python Random Forest model), sending the population in **batches** ensures efficient memory usage and reduces transfer overhead.
* Coalesced memory access patterns ensure that GPU threads access contiguous memory, which maximizes memory bandwidth.

## **Why These Parts Were Parallelized**

1. **Independence**: Population generation, mutation, and crossover are independent for each gene or chromosome, making them ideal for parallel execution.
2. **Scalability**: Parallelization allows the algorithm to handle larger populations and higher numbers of generations without significant performance degradation.
3. **Computational Complexity**: Serial execution of these operations would be too slow for larger datasets and populations. CUDA parallelization drastically reduces the execution time.

## **CUDA Techniques Used**

To optimize the performance of the Genetic Algorithm (GA) using CUDA, we applied several **CUDA-specific optimizations** to maximize GPU utilization, memory efficiency, and computational throughput. Below are the key techniques used:

### **1-Occupancy Maximization**

**What It Means**: Maximizing the number of active threads running on a GPU to ensure efficient use of GPU cores.

**How We Achieved It**:

* We used **256 threads per block** as it is a well-known optimal configuration for many CUDA applications, balancing register usage and shared memory.
* The grid size is dynamically calculated based on the population size and chromosome length to ensure the GPU is fully utilized.

### **2-Efficient Memory Usage**

Efficient memory management is critical to achieving high GPU performance. We used the following strategies:

#### **a) Coalesced Memory Access**

**What It Means**: Ensuring that threads access global memory in aligned and contiguous patterns to reduce memory latency.

**How We Implemented It**:

* Chromosomes (population) were stored in **global memory** in a **contiguous 2D array**, where each row corresponds to a chromosome and columns represent genes.
* Threads in a warp accessed memory sequentially, enabling coalesced memory access.

**Benefit**:

* Reduced memory access latency and improved memory bandwidth utilization during population transfer and fitness evaluation.

#### **b) Batch Processing for Fitness Evaluation**

**What It Means**: Sending data in **batches** instead of individual transfers to reduce overhead from CPU-GPU memory transfers.

**How We Implemented It**:

* Chromosomes were transferred in **batches** as contiguous blocks of memory, leveraging coalesced memory access.
* The fitness evaluation process operated on the batch to minimize I/O delays and improve data throughput.

**Benefit**:

* Improved efficiency of memory transfers, reducing the bottleneck caused by frequent CPU-GPU communication.

### **4. Random Number Generation with** curand

**What It Means**: The **curand** library generates random numbers in parallel for operations like population initialization and mutation.

**How We Used It**:

* Each thread initializes its **curandState** to generate independent random values for genes and mutation probabilities.
* State initialization ensures reproducibility and avoids bias in random values.

### **5. Kernel Fusion**

**What It Means**: Combining multiple operations into a single kernel to reduce kernel launch overhead and global memory accesses.

**How We Applied It**:

* The **population initialization kernel** combines random generation and writing to global memory in one kernel.
* Similarly, the mutation kernel integrates both condition checks and value flipping in a single execution.

**Benefit**:

* Reduces the cost of multiple kernel launches and minimizes global memory latency.

#### **5. Performance Comparison**

## **Serial vs. Parallel:**

The comparison between the serial and CUDA-parallelized implementations of the Genetic Algorithm (GA) reveals significant performance improvements in execution time when leveraging GPU acceleration. The table below summarizes the execution times and speedup achieved:

|  |  |  |  |
| --- | --- | --- | --- |
| Population Size | Serial Execution Time (s) | Parallel Execution Time (s) | Speedup |
| 50 | 546.33 | 54.44 | 10.035 |
| 100 | 903.03 | 99.40 | 9.08 |
| 200 | N/A(Took so long) | 197.0 | N/A |
| 400 | N/A | 383.8 | N/A |
| 1000 | N/A | 941.99 | N/A |

N/A because the execution time of the serial code was so long.

For smaller population sizes (50 and 100), the parallel version achieved a **speedup of ~10x**, drastically reducing execution time compared to the serial implementation.

As the population size increased, the serial code became impractical, and execution times were no longer measurable due to excessive delays. The parallelized version, however, efficiently handled larger populations (up to **1000 chromosomes**), demonstrating the scalability of the CUDA implementation.

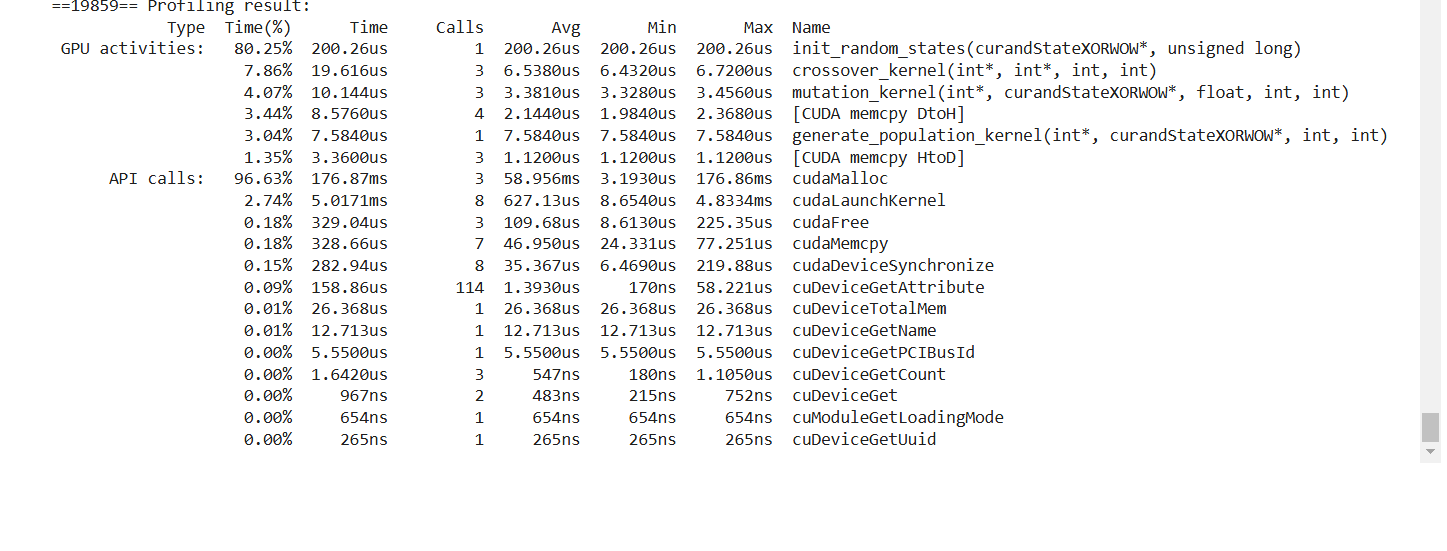
### **Solution Quality**

The **accuracy** of the final solution was consistent for both the serial and parallel implementations, demonstrating that CUDA acceleration did not compromise the quality of the results:

* **Accuracy Achieved**: 0.9924
* **Optimal Selected Features**: [0, 1, 2, 3, 4, 6]

The parallel implementation successfully found the same optimal feature subset as the serial version. These features were later used to train and deploy the final **Random Forest recommendation model**, further validating the reliability of the parallelized solution.

**CUDA Performance Metrics**:



### **Explanation**:

GPU Activities:

**init\_random\_states** (200.26us):

* This kernel takes the most time (80.25%), initializing random states using curandStateXORWOW.
* Likely initializing PRNGs for other kernels.

**crossover\_kernel (7.86%) and mutation\_kernel (4.07%)**:

* These are computational kernels, probably performing genetic operations in parallel.
* Both are lightweight, running in microseconds.

**Memory Operations**:

* **DtoH (Device-to-Host)**: Transfers data from GPU to CPU.
* **HtoD (Host-to-Device)**: Transfers data from CPU to GPU.
* These are expected during execution and can indicate data transfer bottlenecks.

**Other Kernel**:

* generate\_population\_kernel (3.04%): Another computational kernel.

API Calls:

**cudaMalloc (96.63%)**:

* Takes the most API call time, allocating memory on the GPU.
* Memory allocation can be costly if done multiple times.

**cudaLaunchKernel (2.74%)**:

* Launches kernels onto the GPU. The time includes CPU-to-GPU coordination overhead.

**cudaMemcpy (0.18%)**:

* Memory transfer between host and device. It’s reasonably fast here.

**cudaDeviceSynchronize**:

* Ensures all GPU tasks finish before proceeding.

### **Key Insights**:

**Performance Bottleneck**: The init\_random\_states kernel consumes the most execution time. Check if it can be optimized or parallelized further.

**API Overhead**: Most time is spent on cudaMalloc. Consider optimizing memory allocation (e.g., allocating once and reusing memory).

**Memory Transfers**: Device-to-host and host-to-device memory transfers take little time but can add overhead with large data sizes.

**Kernel Execution**: Computational kernels (crossover\_kernel, mutation\_kernel) are fast, but profiling can help further optimize.

# 6-Conclusion:

### **Summary of Findings**

The project successfully implemented a **Genetic Algorithm** for feature selection, comparing both the **serial** and **parallelized CUDA versions**. The accuracy of the model remained consistent at **0.9924** for both implementations, validating the reliability and correctness of the feature selection process. The optimal features selected were: **[0, 1, 2, 3, 4, 6]**.

The significant improvement observed was in **execution time**:

* The serial version faced considerable delays when handling larger population sizes (50).
* The CUDA-parallelized version, however, achieved substantial speedup and efficiently handled a big population size.

The final selected features were later used to **train and deploy a Random Forest recommendation model**, improving performance and reducing computational complexity.

A link has been provided (attached) where this optimized model was integrated and applied effectively.

Link: <https://croprecommendation-fask-2.onrender.com/>

It may take some time for the page to show up since we used the free version of the server.

## **Challenges and Limitations**:

Several challenges were encountered during the project, particularly in debugging CUDA kernels and effectively managing GPU memory. The integration of host-side operations, such as fitness evaluation using external Python scripts, introduced bottlenecks that limited overall performance. Additionally, fine-tuning kernel parameters and achieving optimal parallelization required extensive testing and iteration.

## **Future Work:**

Future improvements could focus on transitioning fitness evaluation to the device, eliminating the latency caused by frequent data transfers between the host and the device. Other optimizations could include leveraging shared memory and streams in CUDA to further enhance kernel performance. Expanding the scope of the genetic algorithm to solve real-world optimization problems or extending it to handle multi-objective scenarios are potential directions for advancing the project.

# 7-Project Management

## **Team Roles and Contributions:**

* **Abbas, Ikram, and Omar** Responsible for overseeing the entire project, ensuring tasks are completed on time, and managing team coordination. We also contributed to defining the project objectives and ensuring proper documentation.
* **Ikram:** Focused on implementing the genetic algorithm (GA) in serial form with the model. She handled the core algorithm development and debugging.
* **Omar:** Optimized the serial code with CUDA for parallelization. Also, verified the correctness and efficiency of the optimized solution through performance tests.
* **Abbas:** Managed the project documentation, including report writing and final presentation preparation. Abbas also ensured that the project's progress was well-documented and that the final report clearly explained the methods, results, and conclusions.
* **Communication Methods:**

We primarily created a **WhatsApp group** for daily communication, sharing updates and quick questions. For more in-depth discussions, **Zoom meetings** were held weekly to review progress, discuss challenges, and plan upcoming tasks. Communication was frequent, with daily updates via WhatsApp and weekly meetings via Zoom to ensure alignment on project goals.

## **Project Timeline:**

* **Week 1:** Initial task selection, project planning, and defining the scope of the genetic algorithm and CUDA optimization.
* **Week 2:** Serial algorithm implementation. The algorithm was first developed in a serial form for basic testing and understanding.
* **Week 3:** CUDA optimization. The serial algorithm was adapted to utilize GPU resources, significantly improving performance for large-scale optimization tasks.
* **Week 4:** Testing and benchmarking. The CUDA-optimized GA was tested on multiple datasets, and its performance was benchmarked against the serial implementation.
* **Week 5:** Report writing and final presentation preparation. The team compiled their findings, results, and conclusions into a detailed report and prepared a presentation to showcase the project's success.

# **References:**

* 1. **Holland, J. H. (1975).** Adaptation in Natural and Artificial Systems. University of Michigan Press.
  2. **Mitchell, M. (1998).** An Introduction to Genetic Algorithms. MIT Press.
  3. **Nvidia CUDA Toolkit Documentation. (2024).** CUDA C Programming Guide. Nvidia.
  4. **Deb, K. (2001).** Multi-Objective Optimization Using Evolutionary Algorithms. Wiley.
  5. **Goldberg, D. E. (1989).** Genetic Algorithms in Search, Optimization, and Machine
  6. **CUDA by Example: An Introduction to General-Purpose GPU Programming**
  7. **CUDA Programming Forum** (<https://forums.developer.nvidia.com/c/gpu-programming-and-performance/cuda/>).
  8. **Stack Overflow** (<https://stackoverflow.com/>).
  9. **CHATGPT**