GA using CUDA Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

Aswini. J

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- Introduction
- In the field of optimization, Genetic Algorithms (GAs) are a powerful class of evolutionary algorithms inspired by the principles of natural selection. They are particularly effective in solving complex problems where traditional optimization techniques may struggle. This project aims to implement a CUDA-accelerated Genetic Algorithm to optimize high-dimensional functions efficiently, leveraging the parallel processing capabilities of modern GPUs.

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- **The workflow of this project can be summarized in the following steps:**

- **Initialization**
- **Fitness Evaluation**
- **Selection**
- **Crossover**
- **Mutation**
- **Iteration**
- **Results Analysis**

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- The workflow of this project can be summarized in the following steps:
- Initialization: A population of candidate solutions (individuals) is generated randomly within a defined search space. Each individual represents a potential solution to the optimization problem. The size of the population and the number of dimensions for each individual can be adjusted based on the complexity of the problem.
- Fitness Evaluation: The fitness of each individual in the population is evaluated using a fitness function. In this project, a high-dimensional sphere function is utilized as the optimization objective, where the goal is to minimize the function value. This evaluation process is performed in parallel using CUDA kernels, allowing for significant speedup by leveraging the GPU's computational power.
- .

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- Selection: Individuals are selected from the population based on their fitness values. The selection process favors better-performing individuals, ensuring that their genetic information is passed to the next generation. This mimics natural selection, where the fittest individuals are more likely to reproduce.

- Crossover: The crossover operator combines pairs of selected individuals to create new offspring. This process introduces genetic diversity and allows for the exploration of new regions in the search space. The crossover rate can be adjusted to balance exploration and exploitation in the optimization process.

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- Mutation: To further enhance genetic diversity and prevent premature convergence, mutation is applied to the offspring. Random changes are introduced to some individuals, ensuring a broader search of the solution space. The mutation rate can be fine-tuned based on the problem characteristics.

- Iteration: The process of evaluating fitness, selecting individuals, performing crossover, and mutating continues over multiple generations. Each generation aims to produce a population with better fitness values than the previous one. The best fitness values across generations are recorded for analysis.

- Results Analysis: The performance of the Genetic Algorithm is evaluated by plotting the best fitness values over generations. Insights are drawn from the visualizations to assess convergence behavior and optimization efficiency. Parameter adjustments can be made to improve the algorithm’s performance based on these insights.

Leveraging-CUDA-for-Efficient-Genetic-Algorithms-in-High-Dimensional-Spaces

- By utilizing CUDA for parallel processing, this project aims to enhance the speed and efficiency of the Genetic Algorithm, making it suitable for tackling high-dimensional optimization problems. This approach not only demonstrates the capabilities of GPU acceleration but also provides a foundation for further exploration into more complex optimization challenges.

Parameter Used Initially

- int main() {
- const int population_size = 1024;
- const int dimensions = 100; // High-dimensional space
- const int generations = 1000;
- const float crossover_rate = 0.7f;
- const float mutation_rate = 0.01f;

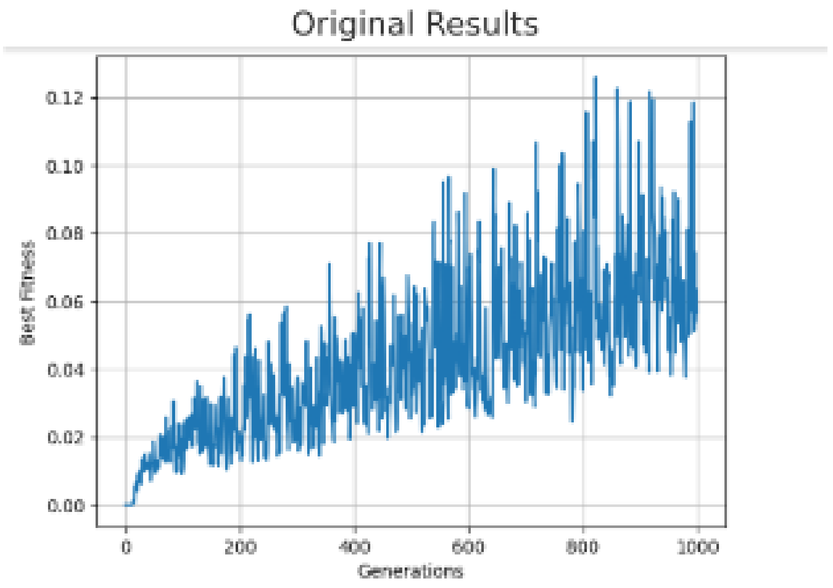
- // Remaining code unchanged...
- }

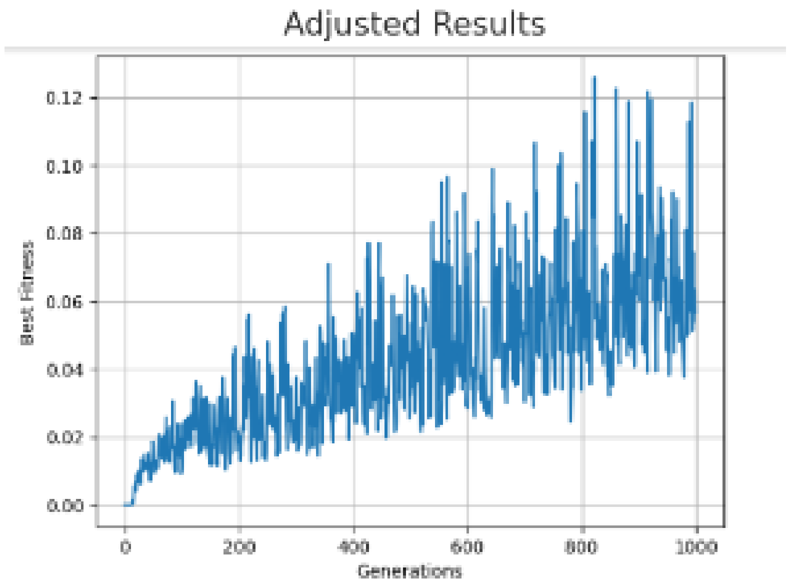
Parameter Adjusted

- int main() {
-     // Adjusted parameters
-     const int population_size = 2048; // Increased population size
-     const int dimensions = 100;   // Number of dimensions (remains unchanged)
-     const int generations = 1500;   // Increased generations for longer runtime
-     const float crossover_rate = 0.8f; // Increased crossover rate
-     const float mutation_rate = 0.02f; // Slightly increased mutation rate

-     // Remaining code unchanged...
- }

Results Obtained





CUDA with shared memory
Population size 1024

CUDA without shared memory
Population size 2048

Conclusion

- The comparison between original and adjusted results shows that both exhibit an overall increase in fitness values over generations, indicating improved solutions.
- However, the adjusted results demonstrate less fluctuation, suggesting smoother convergence and enhanced stability.
- While both sets of results start near zero, the adjusted values are confined to a narrower range of higher fitness, indicating a more focused search.
- Ultimately, the parameter tuning has positively impacted performance, though further adjustments may still be necessary to refine convergence.