Taichi Based GPU Accelerated Evolutionary Algorithm

Ali Asghar Chakera
Department of Computer Science
Habib University
Karachi, Pakistan
ay06993@st.habib.edu.pk

Mustafa Sohail
Department of Computer Science
Habib University
Karachi, Pakistan
ms06860@st.habib.edu.pk

Muhammad Mobeen Movania
Department of Computer Science
Habib University
Karachi, Pakistan
mobeen.movania@sse.habib.edu.pk

***Abstract*—Evolutionary algorithms (EAs) have gained significant attention as powerful optimization techniques across various domains due to their ability to efficiently explore solution spaces and find near-optimal solutions. However, the computational demands of EAs, especially for large-scale optimization problems, have prompted the exploration of parallel and GPU-accelerated implementations to enhance their performance. In this paper, we introduce a novel approach leveraging Taichi, a high-performance computing language specifically designed for GPU acceleration, to develop a GPU-accelerated evolutionary algorithm.**

***Index Terms*—Taichi, GPU, evolutionary algorithms, optimization**

I. Introduction

Evolutionary algorithms (EAs) are a class of computational techniques inspired by the mechanisms of natural evolution, such as selection, mutation, and crossover. These algorithms have found widespread application in solving complex optimization problems, where traditional methods may struggle due to the complexity and scale of the solution space. The core concept behind EAs is the iterative generation of candidate solutions, which are evaluated based on a predefined fitness function. This function assesses the quality or "fitness" of each solution, guiding the evolutionary process toward optimal or near-optimal outcomes.

As each iteration requires evaluating the fitness of multiple candidates, the computational cost can escalate rapidly, especially for problems with high dimensionality or complex evaluation functions. This computational intensity can limit the scalability and efficiency of EAs, particularly when applied to large-scale or time-sensitive optimization tasks. To address this challenge, parallel computing has emerged as a powerful tool. By distributing the computational workload across multiple processors or computing nodes, parallel computing can significantly reduce the time required to execute evolutionary algorithms.

Recent years have seen an increase in the intersection of evolutionary algorithms and parallel computing which has opened new avenues for innovation. This intersectionality has not only accelerated the performance of EAs but also expanded their applicability to a broader range of optimization challenges, including real-time applications and large-scale problems.

This paper focuses on the development of a framework for Genetic Algorithms (GAs) using Taichi, a state-of-the-art programming language designed specifically for parallel computing. Taichi offers a unique architecture for high-performance computation which allows developers to write code that can be efficiently executed across CPUs, GPUs, and other parallel processing units. The main contribution of this paper is introducing a technique for GPU accelerated EAs based on Taichi.

The paper is divided into five sections, with section 2 focusing on related literature, section 3 discussing the methodologies, section 4 highlighting the results, and section 5 discussing the paper in its entirety.

II. Literature Review

The following section will cover the related literature in relation to the topic of the paper. Literature review is divided in two sections with first solely focusing on parallelism in GAs, section two discussing work that used Taichi to optimize EAs in order for a bench-mark to be set.

A. Parallelism in Genetic Algorithms

GAs are based on natural selection and genetics. They simulate evolution by iteratively evolving a population of candidate solutions to find the best outcomes. A GA relies on a population of individuals, or chromosomes, a fitness function to evaluate them, and operators for selection and mutation.

During a GA's execution, multiple chromosomes are evolved and evaluated simultaneously, presenting an opportunity to use parallel computing to speed up the process. With parallel architecture, each chromosome can be generated and evaluated independently, allowing exploration of the solution space in parallel. Synchronization is needed only during the selection and crossover phases, when the entire population is evaluated and knowledge sharing occurs.

According to Hart et al [1], Parallel Genetic Algorithms (PGAs) have the ability to fend off premature convergence which makes them an ideal solution technique. This is on top of the fact that PGAs reduce time to locate a solution and improve the quality of the solution when compared to GAs. The technique used by Alba et al in [2] uses many elementary GAs performing the reproductive plan on their own sub-populations. The number of elementary GAs is another free parameter for Alba et al which they use to their own advantage.

CUDA has been used to solve optimization problems using parallel computing. Cekmez et al [3] uses nVidia's CUDA Random Number Generation Library (cuRAND) using GPU's available cores to maximize the parallel throughput. Additionally, each chromosome is handled in parallel using CUDA. Findings from [3] reported that the parallel model solved the optimization problem in 1777 times faster thus highlighting the importance of parallelization. Huang et al [4] aims to efficiently solve the scheduling problem using the parallelization provided by CUDA. The results from the paper highlighted that the parallel approach was 19 times faster on 200 jobs for the scheduling algorithm.

Attempts on parallelization of the evolutionary process using CUDA has been successful in the past when tested on a range of different optimization problems such as TSP, JSSP, or knapsack.

B. Taichi

Taichi, developed at the CSAIL lab at the Massachusetts Institute of Technology (MIT), is a Python-embedded programming language designed for high-performance parallel computing. It provides significant performance boosts both during development and at runtime. Taichi has been used in the past for computer graphics and image segmentation purposes. The original paper on Taichi [5] highlights how when the performance is compute-bound, our access optimizer can greatly improve performance by reducing access instruction.

Taichi is particularly well-suited for accelerating Genetic Algorithms (GAs) due to its emphasis on parallelism and efficiency. It merges the simplicity and flexibility of Python with the speed and computational power of low-level parallel programming. One of its key features is its ability to seamlessly offload computations to GPUs, enabling large-scale parallel processing.

The intuitive integration with Python is a major advantage, as Python is widely used and familiar to many developers. This seamless integration allows Python users to harness the power of Taichi without needing to learn a new syntax or undergo extensive retraining. As a result, developers can quickly prototype and deploy GA simulations, gaining the benefits of high performance with minimal additional effort.

However, owing to the fact that Taichi language is relatively new, literature and work carried out on the language was sparse. To the best of the research teams capabilities, we were unable to find parallelization techniques of EAs using Taichi.

III. Methodology

The methodology section will cover the implementation details of the experiment. The section is divided into two parts, the first part will cover the implementation of the GA using Taichi and the second part will cover the implementation of the TSP problem.

A. Genetic Algorithm

The Genetic Algorithm (GA) follows the standard evolutionary process, with the following components:

1) Population Initialization: The GA starts by initializing a population of candidate solutions. Each solution is represented as a chromosome, which encodes a potential solution to the optimization problem. The population size is a configurable parameter that influences the diversity of the solutions explored.

2) Fitness Evaluation: The fitness of each chromosome is evaluated using a predefined fitness function. This function assesses the quality of the solution encoded by the chromosome, guiding the evolutionary process toward optimal or near-optimal solutions.

3) Selection: A selection mechanism is used to choose parent chromosomes for reproduction based on their fitness. Higher fitness chromosomes are more likely to be selected, mimicking the natural selection process. There are various selection strategies, such as roulette wheel selection, tournament selection, or rank selection, that can be used to guide the parent selection process. The selected parent chromosomes are then used to create offspring through crossover and mutation for the next generation.

4) Crossover: The selected parent chromosomes undergo crossover, where genetic information is exchanged to create new offspring. This process introduces diversity into the population, allowing exploration of the solution space.

5) Mutation: To further enhance diversity, mutation is applied to the offspring chromosomes. This process introduces random changes to the genetic information, preventing premature convergence and promoting exploration.

6) Replacement: The offspring chromosomes replace the least fit members of the population, ensuring that the population evolves over time. This process repeats for a specified number of generations or until a termination condition is met.

B. Traveling Salesman Problem (TSP)

The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem that involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city. The objective is to minimize the total distance traveled. The TSP is a challenging problem due to its factorial complexity, making it an ideal candidate for testing the efficacy of evolutionary algorithms. The dataset used for the TSP experiments contains the coordinates of the cities.

Dataset: $[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$

1) Problem Representation: In the context of the TSP, each chromosome represents a potential solution to the problem. The chromosome encodes the order in which cities are visited, forming a tour that starts and ends at the same city. The fitness of the chromosome is determined by the total distance traveled along the tour.

Chromosome: $[1, 2, 3, 4, \ldots, n]$

Here, the numbers represent the index of the cities in the dataset, indicating the order in which they are visited in the tour.

2) Fitness Function: The fitness function for the TSP evaluates the total distance traveled along the tour encoded by the chromosome. The distance between each pair of adjacent cities is calculated using a distance matrix, which contains the distances between all pairs of cities. The fitness of the chromosome is the sum of the distances between consecutive cities in the tour.

$$\textbf{Fitness: } f = \sum_{i=1}^{n-1} d(i, i+1) + d(n, 1)$$

Where $d(i, j)$ represents the distance between city i and city j in the dataset.

3) Selection: The selection process in the TSP GA involves choosing parent chromosomes based on their fitness. In the context of the TSP, the selection mechanism aims to favor chromosomes with shorter tour lengths, as they represent better solutions to the problem. In the experiments, we used truncation selection to choose the parent chromosomes. Truncation selection selects the top k chromosomes from the population based on their fitness, where k depends on the offspring population size.

4) Crossover and Mutation: The crossover and mutation operators for the TSP are designed to preserve the validity of the tour representation. Crossover involves exchanging genetic information between parent chromosomes to create offspring, while mutation introduces random changes to the tour. Both operators are implemented to maintain the integrity of the tour representation and ensure that the resulting solutions are valid TSP tours.

In this research, we used point crossover and swap mutation as the genetic operators for the TSP. Point crossover involves selecting a random point in the chromosome and exchanging the genetic information beyond that point between the parent chromosomes.

$$\textbf{Parent 1: } [3, 1, 4, 2, 5]$$

$$\textbf{Parent 2: } [5, 2, 1, 4, 3]$$

$$\textbf{Offspring 1: } [3, 1, 4, 5, 2]$$

$$\textbf{Offspring 2: } [5, 2, 4, 3, 1]$$

To maintain the validity of the tour representation, the offspring chromosomes are then repaired using a repair operator that ensures that each city is visited exactly once in the tour. The repair operator loops through the the parent chromosome until an unvisited city is found and adds it to the offspring.

Swap mutation involves selecting two random points in the chromosome and swapping the cities at those points. This introduces random changes to the tour while maintaining the validity of the solution.

$$\textbf{Chromosome: } [3, 1, 4, 2, 5]$$

$$\textbf{Mutated Chromosome: } [3, 4, 1, 2, 5]$$

The mutation and crossover operators are applied iteratively to the population of chromosomes, guiding the evolutionary process toward optimal or near-optimal solutions to the TSP.

C. Implementation using Taichi

The Genetic Algorithm (GA) for the Traveling Salesman Problem (TSP) was implemented using Taichi, a high-performance computing language designed for parallel processing. Since Taichi lang is built on Python, the implementation required minimal changes to the existing GA code, making it easy to leverage the power of GPUs for accelerating the evolutionary process. The implementation consisted of the following steps:

1) Population Initialization: The population of chromosomes was initialized using Taichi's data structures and parallel computing capabilities. As Taichi leverages GPU for parallel processing, it requires defining the data structures and operations in a way that can be efficiently executed on the GPU. Taichi supports 2D arrays which it converts to linear arrays for GPU processing. The population of chromosomes was represented as a 2D array, where each row corresponds to a chromosome.

```
population = ti.field(dtype=ti.int32, shape
    =(population_size, chromosome_size))
@ti.kernel
def init_population():
   for i in range(population_size):
      for j in range(chromosome_size):
         population[i, j] = j + 1
      shuffle(i)
      calculate_fitness(i)
```

The population was initialized by assigning each chromosome a unique order of cities and calculating its fitness based on the TSP fitness function. The shuffle function randomly permutes the order of cities in the chromosome to introduce diversity in the initial population.

Taichi provides a kernel decorator to define parallel kernels that can be executed on the GPU. The outer loop in the init_population kernel is executed in parallel, with each thread handling a different chromosome in the population.

IV. Experimentation and Results

A. System Configuration

The system used for the experiments was a desktop computer with the following specifications:

1) CPU Specifications:

- Intel Core i7-9700K @ 3.60GHz
- 8 Cores, 8 Threads
- 64 GB DDR4 2133 MHz RAM

2) GPU Specifications:

- NVIDIA RTX Titan
- 4608 CUDA Cores
- 24 GB GDDR6 VRAM
- 384-bit Memory Bus

B. Experimental Setup

The experiments were conducted using the qa194 TSP benchmark dataset, which contains the coordinates of 194 cities in Qatar. The goal was to optimize the runtime performance of the GA using Taichi and leverage the GPU acceleration to speed up the evolutionary process.

C. Results

The results of the experiments demonstrated the effectiveness of the GPU accelerated GA implemented using Taichi. The runtime performance of the GA was significantly improved by leveraging the parallel computing capabilities of the GPU.

V. Future Work and Conclusion

VI. Bibliography

1) Hart, W. E., Baden, S., Belew, R. K., & Kohn, S. (1996, April). Analysis of the numerical effects of parallelism on a parallel genetic algorithm. In Proceedings of International Conference on Parallel Processing (pp. 606-612). IEEE.
2) Alba, E., & Troya, J. M. (2002). Improving flexibility and efficiency by adding parallelism to genetic algorithms. Statistics and Computing, 12(2), 91-114.
3) Cekmez, U., Ozsiginan, M., & Sahingoz, O. K. (2013, November). Adapting the GA approach to solve Traveling Salesman Problems on CUDA architecture. In 2013 IEEE 14th International Symposium on Computational Intelligence and Informatics (CINTI) (pp. 423-428). IEEE.
4) Huang, C. S., Huang, Y. C., & Lai, P. J. (2012). Modified genetic algorithms for solving fuzzy flow shop scheduling problems and their implementation with CUDA. Expert Systems with Applications, 39(5), 4999-5005.
5) Hu, Y., Li, T. M., Anderson, L., Ragan-Kelley, J., & Durand, F. (2019). Taichi: a language for high-performance computation on spatially sparse data structures. ACM Transactions on Graphics (TOG), 38(6), 1-16.