UNIVERSITÀ
DI SIENA
1240

High Performance Computer Architecture
Assignment

Parallel Genetic Algorithm to solve the TSP

Pietro Pianigiani

Academic Year 2023-2024

Project Abstract

Genetic Algorithms, inspired by natural selection mechanisms, set the grounds for the solution of hard optimization problems removing the necessity to employ strong constraints but focusing on the search of solution space using an unbiased but directed approach. Some evolution environments often applied, strongly rely on the independence of subgroups of individuals, opening the possibility to distribute computational loads on multiple processing units. Such facility enables a growth in the size of the population employed in genetic selection, leading, as proven in literature [1], to a more efficient exploration of the solution space.

In the context of this project we delve into the implementation of a Genetic Algorithm based on the Island model and seek to parallelize its execution both on Multicore processors and on GPUs using multithreading and the CUDA platform. The aim is to prove the effectiveness of the parallel approach when dealing with an increasing number of individuals, measuring execution times and overall performances on the Traveling Salesman NP-hard optimization problem.

Contents

Chapter 1

Introduction

1.1 Genetic Algorithm

Genetic Algorithms (GAs), a subset of Evolutionary Algorithms (EAs), are meta-heuristic techniques extensively utilized in operations research. They excel in finding optimal or near-optimal solutions to complex optimization problems characterized by non-polynomial time complexity. By mimicking natural evolutionary processes such as selection, mutation, and crossover, GAs efficiently explore the solution space for diverse problems. This behaviour is achieved through the modeling of the mentioned solutions in the form of individuals, often referred to as chromosomes.

This shifts the modeling of the optimization solver from the characteristics of the problem to the biological representation of the individuals, providing a typically easier setup for general adaptation of the GA.

1.2 Island Model

Referred to as the "Island model", this paradigm is inspired by the idea of isolated evolution of sub-groups of individuals in different environments. The initial set of randomly proposed solutions is split in equally sized populations whose only interaction with the others is reduced to a periodical migration of the fittest members to neighbouring islands. This paradigm does not only promote diversification of individuals, given varying characteristics of the environment, but practically isolates the required computational load of evolution to the single islands, allowing for parallelization on multiple computing units. A brief sketch of a generic Island model is shown in Fig. 1.1

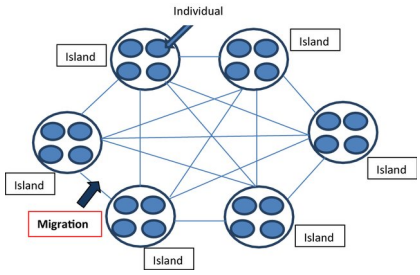

Figure 1.1: A generic example of Island model of a population.

1.3 Genetic Algorithm steps

Decided the characterization of the single chromosomes, the typical procedure followed by the Genetic Algorithm can be summarized as follows:

1. **Initialization:**
   A batch of N new random individuals is created.

2. **Evaluation:**
   The individuals are sorted based on a decided Fitness Criterion.

3. **Selection:**
   A portion of the fittest individuals is selected to produce offspring for the next generation

4. **Crossover:**
   According to the decided strategy, a number of parent chromosomes are used to produce an equivalent number of children individuals.

5. **Mutation:**
   Random or heuristic modifications are applied to part of the genomic structure of the offspring generation.

6. **Stopping criterion:**
   If the stopping criterion is not met, the process is repeated from point 2

In the particular case of the Island Model, a new procedure named **Migration** is typically added after the Mutation operator to let the fittest individuals of each environment attempt to copy themselves on one of the other islands (In case of success, the least fit individuals are overwritten with the new genomes).
Each step of the algorithm can be characterized by the employment of specific heuristics depending on the domain knowledge of the problem.

1.4 Exploring the Solution Space

Evolutionary Algorithms can be shortly described as techniques to explore the space of feasible solutions for a given problem in an efficient manner. Relying on natural selection mechanisms, the Genetic Algorithm is capable of evolving a fixed in size set of individuals towards fitter genomic structures. Nonetheless, such procedure can be expensive in terms of time and prove unstable due to the randomness introduced by initialization and mutation operator.
The literature [1] proves a positive correlation between stability and time of convergence and the size of the population posed to the evolution process. Though intuitively functional, such growth in the number of considered individuals, easily results in longer computational times. Relying on the highly parallel structure of the Island Model, a Divide et Impera approach can be used to tackle the computation.

1.5 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a well-known NP-hard optimization problem consisting in the search of the shortest possible path traversing uniquely a given set of nodes or cities and returning to the start. Various formulations and different constraints such as higher dimensionality of the considered space and the absence of bidirectional paths between certain cities, can be considered depending on the desired complexity.

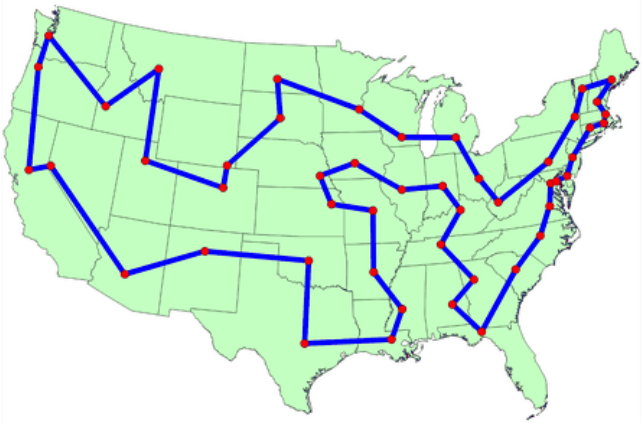

Figure 1.2: `att48` dataset, symmetric traverse order of 48 states in US, *Wikipedia* [2]

Known exact solutions to the TSP reduce the required complexity from the *bruteforce* procedure, consisting of the comparison of all the possible permutations of nodes ($O(N!)$), to exponential forms, with the best (quantum) algorithm found executing in $O(1.728^N)$ where N represents the number of considered cities.
Such computationally expensive problems can be partially solved relaxing the requirements to sub-optimal solutions, one of the commonly used techniques for TSP is the Genetic Algorithm.

Chapter 2

Genetic Algorithm to solve TSP

2.1 Benchmark

In the specific scope of this project, the 2D Euclidean and Symmetric version of Traveling Salesman Problem was considered as the benchmark, on which both accuracy and, most importantly, execution times of the proposed Genetic Algorithm solver were tested.

2.2 Data Representation

TSP datasets can be mainly divided in three different data formats [3]:

- **List of N-dimensional Coordinates**: This format uses coordinates to define the locations of cities in an N-dimensional space. It is ideal for problems where the physical layout and accurate distance calculations are necessary, using geometric formulas like the Euclidean distance.
- **Distance Matrix**: This format provides a matrix where each entry (i, j) represents the distance from city i to city j. This precomputed matrix is especially useful when the exact distances are known beforehand, allowing for faster access during the algorithm's execution, which can be crucial for computational efficiency.
- **Labelled Weighted Graph**: In this graph-based format, cities are vertices and paths are edges with weights indicating the travel cost or distance. This format is versatile, supporting variations such as directed edges (asymmetric distances) and is well-suited for algorithms that leverage graph theory.

Our solver takes into account the solution for datasets of the first kind, parsing a spaced list of 2D coordinates and only then producing a euclidean distance matrix used for efficient calculations.

2.3 Chromosome Representation

In the context of Genetic manipulation of solutions for a given problem, a so-called chromosome or individual is the set of single genes composing a feasible solution, namely a

generic non-optimal result that can be evaluated in quality.
A suitable representation often used to solve TSP is that of creating individuals made of unique entries of indices representing the cities with respect to an a priori fixed order. Given a list of cities in coordinates format:

$$C = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{S-1}, y_{S-1})\}$$

Where s denotes the total number of cities, Two different individuals can be characterized as:

$$I_1 = \{0, 1, 2, \ldots, S-1\}$$
$$I_2 = \{6, 5, S-1, \ldots, 0\}$$

Having a fixed size of S unique indices in the range $[0, S-1]$.

2.4 Population Representation

During evolution process, an initial set of size N of individuals is repeatedly evaluated, selected, bred and mutated to get advantage of randomly appeared genetic orderings that improve the total traversing distance. According to the Island Model, the procedure is individually applied to fixed in size sub-groups of chromosomes named Islands. A feasible representation ([4], [5]) for such structure is a matrix containing the entirety of the population divided as follows and shown in 2.1:

- Every row is an individual
- Every group of M rows is an Island

Evolution of each Island is treated separately from the others, allowing in this way both sequential and parallel access.

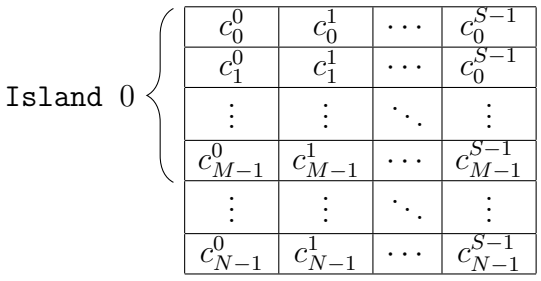

Figure 2.1: Population data structure

2.5 Implementations of GA

For the purposes of this project a Genetic Algorithm was developed onto the paradigms of Single and Multi Core CPU processing and GPU processing.
Due to the structure of the algorithm not all the phases can be subject to complete parallelism and in fact, when used, this can only take place within the single genetic operators of **Selection**, **Crossover**, **Mutation** and **Migration**, with this last in particular forcing the use of a synchronization point within each generation. Nonetheless the most intensive operations can strongly benefit from such strategies.
The *baseline* program is the Single Core version of it consisting of the following distinct phases:

- Load chosen dataset and create the distance matrix
- Create the population
- Shuffle all the chromosomes' genes
- Evaluate the Islands (Sorting individuals on each)
- Evolve the population (Repeat until stop criterion is met):
    1. Select top $x\%$ individuals on each island
    2. Breed top $x\%$ individuals generating offspring and keep absolute best parent
    3. Mutate each newly generated chromosome
    4. Evaluate the Islands (Sorting individuals on each)
    5. Attempt (with certain probability of success) to migrate top $x\%$ individuals to adjacent islands every G generations
- Retrieve the best overall individual (sorting the best chromosomes of each island)

For the parallelization on multiple cores, three attempts were made, starting from the simple deployment of new threads for each sub-group of islands that needed to be recreated on every generation, following with the use of a **ThreadPool** class and finally relying on **OpemMP** facilities.
In order to take advantage of Graphics Card acceleration, one last CUDA version of the code was built with customizable parameters and allowed for the handling of genetic operators on the granularity level of single islands [4]. Further considerations on this last approach are made in the Kernels subsection of Chapter 3.

Chapter 3

Parallelization of Genetic Operators

3.1 Introduction

In this chapter we present the means of parallelization adopted in the two different approaches on a `Multi Core CPU` and on a `GPU`, the main inspiration for the overall code structure comes from [5].

3.2 CPU

Working on the CPU limits the amount of physical workers to the number of cores of the used Processor, the aforementioned genetic operators need to be parallelized on the level of a suitable and evenly divided number of sub-groups of Islands.
For all the iterations of the Multi Core program this measure can be defined as:

$$\lceil \frac{N/M}{\#AvailableCores} \rceil$$

With N and M being respectively the total number of individuals and the number of individuals per island.
In total, three different attempts were made to harness the possible parallelism on a CPU:

3.2.1 Plain Thread Generation

The first attempt consisted of the use of the `std::thread` facility offered by C++ standard library, practically generating and assigning single threads when needed within generations. Though functional such approach strongly relies on the scheduling capability of the OS to cleverly execute the requested workers on the effective cores and introduces brief delays due to the continuous join and generation of new instances of threads.

3.2.2 Threadpool Class

Starting from the codebase of the first version, a new attempt to reduce overhead was made by developing a Threadpool class to reuse uniquely generated threads of execution. Through a series of synchronization mechasims and the use of `std::mutex`

or `std::condition_variable`, the workers are kept running until a task (or function) presents in a shared queue and is picked for execution. In a Linux environment, through the pthreads API, a thread affinity mechanism can also be employed to assure strict execution of the selected threads on physical cores. This is realized with calls to the following function:

```
1  void set_thread_affinity(int core_id) {
2      cpu_set_t cpuset;
3      CPU_ZERO(&cpuset);
4      CPU_SET(core_id, &cpuset);
5      pthread_t current_thread = pthread_self();
6      if (pthread_setaffinity_np(current_thread, sizeof(
           cpu_set_t), &cpuset) != 0) {
7          std::cerr << "Error setting thread affinity for core
               " << core_id << std::endl;
8      }
9  }
```

3.2.3 OpenMP

One last attempt to simplify the code structure was made relying on the `OpenMP` library to relax the means of parallelization of single functions through the use of `#pragma parallel for (num_threads)` calls.
Workers are still pinned to processing units setting specific environment variables such as `OMP_PLACES=cores` and `OMP_PROC_BIND=spread`, with the latter forcing the threads to execute on the largest available number of places, set by the former to the physical cores of the CPU.

3.3 Considerations

All the three iterations produced significant speed-ups on the Genetic Solver for TSP, with the last and more optimized `OpenMP` version performing slightly better than the others. A particular consideration must be made on the means of random numbers generation when operating with `Multi Core` code, specifically, to enforce thread safe operations, instances of `std::mt19937` were used, seeding each with the sum of `system_clock` values and the thread id.
On the tested scale, all the `Multi Core` attempts performed comparably in reducing the time of execution of the Genetic Algorithm.

3.4 GPU

Parallelization on Graphics Cards environment enables, thanks to the much larger availability of computing units, the reduction of the granularity up to the level of single islands, with single cores operating on the M individuals composing the sub-populations. This, as shown in the results, largely benefits the plausible total amount of chromosomes posed

to evolution process and thus the efficient exploration of solution space.
For this purpose, one last version of the codebase was developed through the facilities offered by the Compute Unified Device Architecture (CUDA) Platform.

3.4.1 Resource Limitations

Due to the necessity of handling a large number of individuals, one first decision had to be made. Some of the genetic operators could in fact benefit from lower granularity levels, given the strong independence of data in those contexts, single chromosomes could potentially be handled uniquely by single cores. Nonetheless, this would produce, considering not all the kernels share the property, an increasing demand for computing units with the growth of total population. Due to this assumption, operators such as **Mutation** or **Random Shuffle**, were "inefficiently" scaled up to the handling of islands instead of single individuals.
In practice, considered the finite amount of `Blocks` that can be requested to the GPU, letting the solver capable of handling large numbers of islands means limiting the possibility to leverage data independence within those.

3.4.2 Kernels

A series of sequentially executing Kernels were developed to parallelly tackle individual operations needed for Genetic Evolution, assuming the necessity of high numbers of reuse of those with the growth of total generations, calls to `cudaDeviceSynchronize()` function were employed to enforce the master thread to enqueue fewer requests in the only `Streaming Multiprocessor` used. This mean of synchronization is also necessary to avoid the start of following operators before the completion of all the portions of a certain task. Whenever necessary, to handle randomness an API call to the `curand()` function, preceded by `curand_init()` seeding is used.

In order of appearance the six developed Kernels are:

- **random_shuffle**:
  Shuffles all the genes in each chromosome within an island.
- **calculate_scores**:
  Fills in the sub-portion of Fitness and Distance vectors relative to a single island using the distance matrix.
- **genetic_step**:
  Handles the sorting of individuals on a single island and locally actuates the crossover operator on fittest parents.
- **mutation**:
  Mutates two random genes of each chromosome in a single island.

- **migration**:
  Replace the least fit individuals of the adjacent following island with the top $x\%$ of a single island.

- **fit_sort_subpop**: Similarly to the first part of genetic_step, sorts the individuals on an island based on their fitness scores.

To address the allocation of resources for each Kernel, the performance were measured through the **Nvidia Nsight Profiler**, resulting in a fixed choice for the `grid size` and the `block size`, respectively indicating the number of requested blocks and threads within each block. Selected the `threadsPerBlock` #define variable as a multiple of the warp size, the dimensionalities of grids and blocks can be defined in a 1D fashion as follows:

$$grid = [\lceil \frac{N/M}{threadsPerBlock} \rceil, 1, 1] \ , \ block = [threadsPerBlock, 1, 1]$$

The choice of the `threadsPerBlock` permits a good balance between blocks and threads allocation and demonstrates an important parameter when scaling up the population size.

3.4.3 Considerations

Given the nature of the GPU devices, a notable overhead is obviously introduced when moving data from RAM to VRAM at the start and the end of each run of the program. Nonetheless the overall measured improvement is quickly shown when increasing the population size and proves resilient to amounts of individuals that would be unmanageable on the CPU alone. This fact by itself is sufficient to conclude the possibility for a more efficient exploration of the solution space in a fixed amount of time, when compared to all the previous iterations.
A partial flowchart that shows the degree of parallelism reached on the GPU is shown in Fig. 3.1

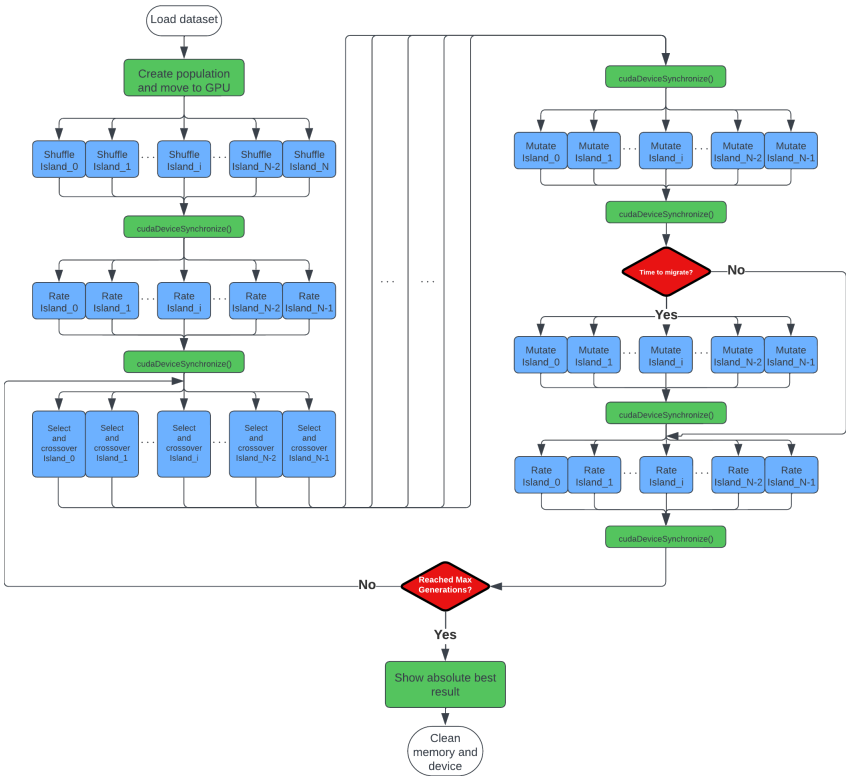

Figure 3.1: Execution of Genetic Algorithm with emphasis on parallel steps.

Chapter 4

Results and Conclusions

In this chapter we present an overall comparison of the results obtained through the parallelization of our Genetic Solver on the TSP. A particular emphasis is put on the absolute advantage gained by the GPU version when compared with a more typical `Single Core` solution. The tests were conducted on two different Processors and GPUs respectively installed in different machines, the specifics of the architectures are mentioned in Appendix B.

4.1 Standardization of the tests

Some considerations on the nature of the tests and the comparability of obtained results is necessary to asses the quality of the work.

4.1.1 Time measurements

Due to the nature of the programming languages used, different means of time measurement had to be taken into account. Particularly, the iterations of the program that didn't take advantage from multi-threading, namely **Single Core** and **CUDA**, were written in C language. To take timestamp checkpoints, the `clock()` function from `time.h` library was employed. This results in a measure of the number of CPU clock ticks elapsed since the start of the program; via a ratio with the global variable `CLOCKS_PER_SEC` this is transformed in a measure of time in milliseconds.

When dealing with parallelization on multiple CPU cores, a thread-safe method needed to be taken into account. For this purpose, every `Multi Core` version relies on instances of `std::chrono::high_resolution_clock`, with calls to the `now()` member function. Through a `duration_cast` this is then converted in milliseconds unit.

Considered the average amount of time spent in the executions, we assume negligible any tiny mismatch in the mean of measurement on this scale that might be due to the use of two different methods.

4.1.2 Code structure

To assure a reasonable comparison between each program, C standard practices were employed in every shared scenario, avoiding C++ facilities when only `Multi Core` versions could take advantage of them. Particularly, dynamic allocation of data was always done through `malloc()` calls and data structures such as matrices and vectors were represented through standard C type arrays.

4.1.3 Population size

To advantage GPU usage without limiting CPU performance, the total number of individuals posed to evolution process is kept fixed to multiples of the warp size, thus exploiting a reasonable number of threads within each requested block in the `Streaming Multiprocessor`.

4.1.4 Hyperparameters and dataset

Throughout the entirety of the tests the set of hyperparameters employed in the Evolution process were kept fixed to equal values. Specifics of such control variables are given in the Appendix B of this work.
The chosen dataset consists of the coordinates values of the `att48` from `TSPLIB`.

4.2 Tests on average generation time

For starter, our tests devised the respective computation time needed to complete a single generation of evolution on a variable number of individuals and thus islands. This allows to move out of the comparison the obvious initial overhead due to the transfer of data towards the GPU device.
To gather a strong measure of the average timings, 5 runs on 50 generations each have been launched for every tested environment.
The charts in Fig. 4.1 and Fig. 4.2 show the analyzed scenarios on the two employed machines.

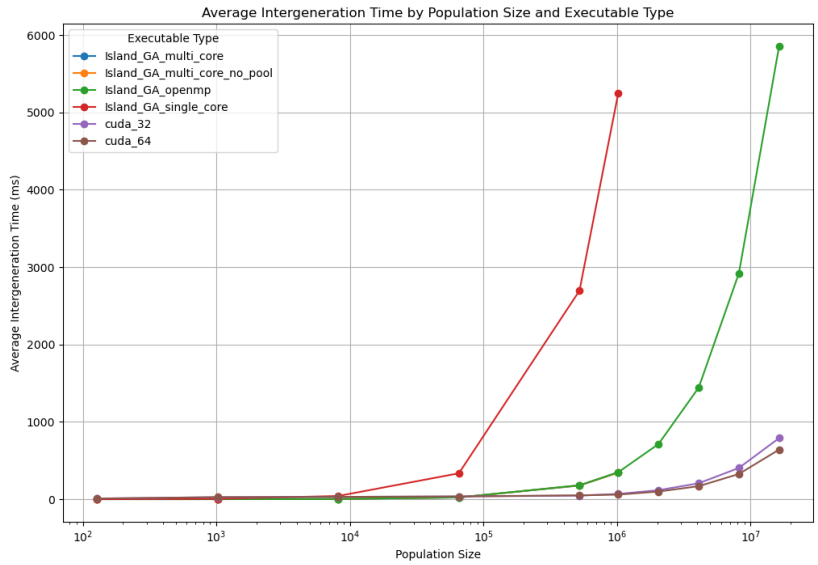

Figure 4.1: Population size vs Average time spent on a single generation - CPU/GPU 1

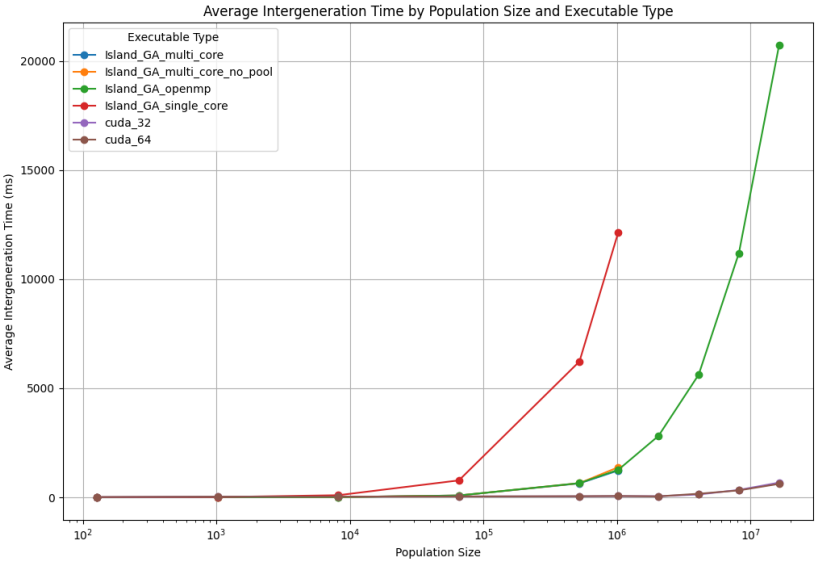

Figure 4.2: Population size vs Average time spent on a single generation - CPU/GPU 2

For a relatively small number of individuals, specifically fewer than 10^4, the timings are consistent across tests. However, both the GPU and Multi Core parallel versions of the code demonstrate greater resilience when managing larger populations, with the GPU version in particular surpassing the performance of all others.
It is worth noting that all iterations of the Multi Core program operate within a similar time-frame at this scale. Consequently, when the population size reached 10^6, the focus

of testing shifted solely to the OpenMP and CUDA versions for reasons of efficiency, completely moving out the too slow Single Core.
While the performance remains comparable, for populations exceeding 10^7 individuals, configuring the system with 64 threads per block (t.p.b.) instead of 32 offers a slight speed advantage.

In Table 4.1, the average recorded times on the first machine are detailed, highlighting the speed-up ratio achieved by the CUDA version with 64 t.p.b. compared to the Single Core configuration.

Table 4.1: Comparison of execution timings for different configurations

Population size	**Single-Core (ms)**	**OpenMP (ms)**	**CUDA (ms)**	**Speed-up ratio**
8192	41.46 ms	20.18 ms	29.81 ms	1.39
65536	334.77 ms	22.75 ms	35.76 ms	9.36
1024000	5251.86 ms	349.16 ms	59.06	88.92
16384000	104013.75 ms	5855.54 ms	640.88 ms	**162.30**

4.3 Tests on average total time

As previously mentioned, operating with GPU devices introduces evident overheads when dealing with data transfers on the bus. Within the scope of our solver, this operations happen at the beginning and the end of the evolution procedure. To assure a consistent gain in the employment of CUDA facilities, a timing test is conducted on the average total time in a 50 generations run. The test is repeated 5 times to safely measure the average delays. After reaching unfeasible amount of time for a Single Core, this is again moved out along with Multi Core versions other than OpenMP and the run reduced to 25 generations.

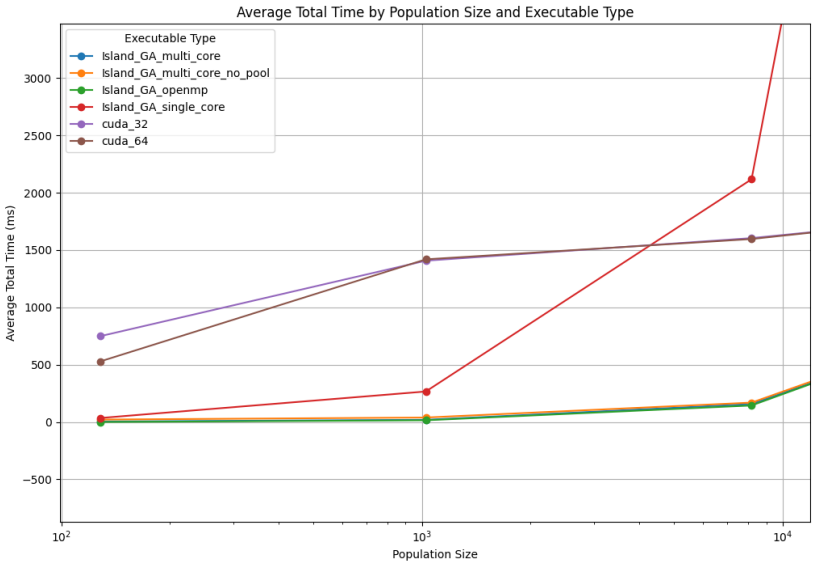

Figure 4.3: Number of individuals vs Average total time - Up to 10^4

As expected, on the low end of the tests, GPU tends to slow the process significantly, due to the unnecessarily moved data, often losing against competitor versions as shown in Fig. 4.3 on the first machine.
This disadvantage is not only recovered with a growth in the number of generations but also when increasing population size on this scale.
Overall results for both the computer systems are shown in Fig. 4.4 and Fig. 4.5.

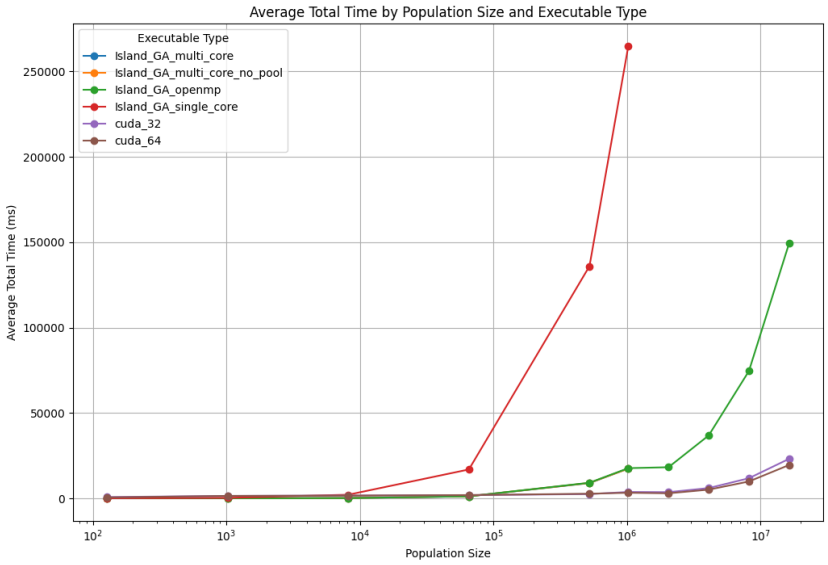

Figure 4.4: Number of individuals vs Average total time - CPU/GPU 1

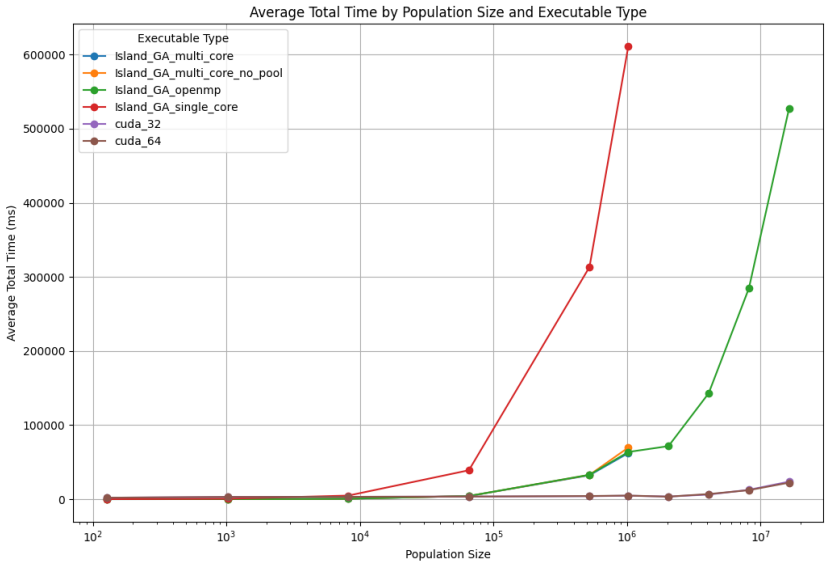

Figure 4.5: Number of individuals vs Average total time - CPU/GPU 2

Again, a notable higher resilience to increasing population size is shown in the CUDA version of the solver, with an approximate speed-up ratio on the second machine of **124.72** against the Single Core with 1024000 chromosomes and of **23.63** against OpenMP with 16384000 chromosomes.
Given a standard run covering around 10.000 evolution steps, even a smaller speed-up can significantly reduce the computation time.

4.4 Tests on quality of solution in fixed time

For the purpose of this project, the TSP was taken in consideration purely as a computational benchmark, nonetheless a quality assessment of possible solutions found with our solver is necessary to validate the overall consistence of the developed application.

Tab. 4.2 details the achieved results on three long runs of 10 minutes on the Single Core, OpenMP and CUDA versions with 512000 individuals, trying to solve the `att48` dataset from `TSPLIB`, with a known optimum of **33522**[5]. A remark on the total number of generations computed in this fixed time frame is highlighted to emphasize the advantage of parallelization.

Table 4.2: Quality of solutions on the different configurations - CPU/GPU 1

Configuration	Solution total distance	# generations	% from optimum
Single Core	41424.86	241	23.57 %
Multi Core	34658.79	1358	3.39 %
CUDA	**34006.61**	**9583**	**1.45 %**

As expected the GPU running solver can squeeze a much greater amount of generations within the 10 minutes, thus reaching a better solution in the average case; it's in fact worth noting that due to the random initialization and decision making of the process no guarantees are offered in reaching optimal or sub-optimal solutions though incrementing evolution epochs.
A graphical representation of the best solution found is shown in Fig. C.1 in the Appendix C.

4.5 Conclusions

All the analyzed scenarios demonstrated the superiority of GPU parallelization compared to other techniques when employing a substantially larger number of individuals. Within the scope of this project, we conclude that the overall benefit of employing CUDA facilities is highly valuable and capable of scaling effectively with problem size. The GPU-based solver not only provided significant speed-ups but also achieved better average solution quality, particularly in large-scale TSP instances. This makes GPU parallelization a compelling choice for leveraging the power of Genetic Algorithm and natural selection mechanisms.

4.5.1 Follow-ups

In this work we explored in relative depth the possibility of reducing overall computation time for our proposed algorithm. Most of the genetic heuristics took strong inspiration from literature works [6] and were then adapted to a manageable level of granularity, this could nonetheless be highly improved with further experimenting with specific architectural constraints and more sophisticated in-place techniques.
The general idea behind the project could also be extended to differently shaped optimization problems considering different requirements and thus benefit from the highly parallel structure of the GA.

Appendix A

Project Organization

In this appendix the overall structure of the project codebase is discussed to allow easier navigation and replication of the tests.

A.1 Code Components and Languages

To allow preliminary testings, various environments and tools were developed in mixed languages, often relying on the ease of use of Python for simple plotting purposes and pseudo code tests. The main components of the codebase are divided as follows:

- `run_test.sh` - Compiles and launches the *baseline* test on all the variations of the solver with averages over 5 runs and 50 generations
- `further_test.sh` - Compiles and launches the *extended* test on all the variations of the solver with averages over 5 runs and 25 generations
- **data/** - Contains all the used datasets and accepts coordinates sequences as .txt files of the format:

$$\begin{matrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{matrix}$$

- **results/** - Contains test results and last execution results along with previous successful runs plots and/or data:
  - last_result.txt - contains the summary for the baseline test conducted on all iterations and launched through `run_test.sh`
  - new_result.txt - contains the summary for the extended test conducted solely on CUDA and OpenMP and launched through `further_test.sh`
  - `plot_avg_gen_time.py` - A script to plot all the average generation times from last_result.txt

- `plot_avg_total_time.py` - A script to plot all the average execution times from last_result.txt
- best_solution.txt - The coordinate list for the last executed solver executable, saved here for plotting purpose

- **src/** - Contains all the three versions of the solver with variations:

  1. `Island_GA_single_core.c`
  2. `Island_GA_multi_core_no_pool.cpp` - Base Multithreaded version
  3. `Island_GA_multi_core.cpp` - Multithreaded version using a Threadpool class
  4. `Island_GA_openmp.cpp` - Multithreaded version using OpenMP
  5. `Island_GA_cuda.cu` - GPU accelerated version written in CUDA

  Contains also:

  - **utils/** - Includes some Python scripts for ease of plotting and testing:
    1. `brutal_solver.py` - Simple $O(n!)$ test to solve short TSP instances
    2. `plot.py` - Plots the last saved result from one of the solver executables
  - **logs/** - Contains all the .log files in testing phase for each executable

Appendix B

Hardware Setup & Software Specifics

B.1 Hyperparameters

Excluding the version specific control variables that regulate the degree of parallelism or requests of resources allocation (CUDA), each iteration of the Genetic Algorithm TSP solver takes into account the same set of environmental and randomness hyperparameters. Here follows a brief description and the assigned values in the testing phase:

- `subPopSize` = 32 - number of individuals inhabiting a single island, previously mentioned in the work as M
- `selectionThreshold` = 50% - percentage of individuals that are subject to crossover to generate new offspring
- `migrationAttemptDelay` = 10 - number of generations before the fittest individuals of an island try to migrate (replicate) on following adjacent island
- `migrationProbability` = 70% - probability that a migration happens successfully (applies to all the islands)
- `migrationNumber` = 4 - number of fittest individuals subject to migration
- $\alpha = 0.2$ - environmental advantage of the island
- $\beta = 0.7$ - base hospitality of the experiment
- $\gamma = 0.2$ - environmental replication disadvantage of the island
- $\delta = 0.3$ - base replication disadvantage of the experiment
- `crossover_rate` = $[\frac{\alpha \times (island\_index+1)}{popSize/subPopSize} + \beta]$ - probability of generating offspring from two selected individuals, stays within $[\beta, \alpha + \beta]$ depending on the island
- `mutation_rate` = $[\frac{\gamma \times (island\_index+1)}{popSize/subPopSize} + \delta]$ - probability of mutating newly generated offspring, stays within $[\delta, \gamma + \delta]$ depending on the island

B.2 Tested Computer Systems

The experiments were conducted on two different machines, namely a high-end mobile MSI laptop and a Google Cloud Compute Engine instance. The specifications of each in hardware and software follows:

First Machine

Component	**Specification**
System	MSI Raider GE78HX
CPU	IntelCore i7-13700HX @ 2.10 GHz **(16 cores, 24 threads)**
CPU Cache	L1 80 KB/Core, L2 24 MB, L3 30 MB
RAM	32GB DDR5 5600MHz
GPU	**NVIDIA GeForce RTX 4070** @ 2.17 GHz
GPU Architecture	Ada Lovelace
GPU RAM	8GB GDDR6X 8001 Mhz
CUDA Capability	8.9

Table B.1: Hardware Specifics of the **first machine**

Software distributions and version:

- Operating System: Ubuntu 22.04.1 LTS
- CUDA Toolkit: 12.3
- C Compiler: GCC Version 11.4.0
- CUDA Compiler: NVCC Version 12.3.107

Second Machine

Component	**Specification**
System	Google Compute Engine
Kernel	5.10.0-29-cloud-amd64 x86_64
Distribution	Debian GNU/Linux 11 (bullseye)
CPU	Intel Xeon (Cascade Lake) @ 2.20 GHz **(4 cores, 8 threads)**
CPU Cache	L1 128 KiB/Core, L2 4 MiB, L3 38.5 MiB
RAM	32GB (details not provided)
GPU	**NVIDIA L4 GPU Accelerator**
GPU Architecture	Ada Lovelace
GPU RAM	24GB GDDR6 6251 Mhz
CUDA Capability	8.9

Table B.2: Hardware Specifics of the **second machine**

Software distributions and version:

- Operating System: Debian GNU/Linux 11 (bullseye)
- CUDA Toolkit: 11.8
- C Compiler: GCC Version 10.2.1
- CUDA Compiler: NVCC Version 11.8.89

Appendix C

Plots & Datasets

This appendix collects briefly a couple of significant plots of obtained results on the two mainly tested datasets from `TSPLIB`, namely the `att48` and the `kroA100` (Krolak A) coordinates sequences. In addition a novel short dataset of 11 cities that have been employed in the preliminary phase of the development.

- Best result on `att48` in 10 minutes and 512000 individuals

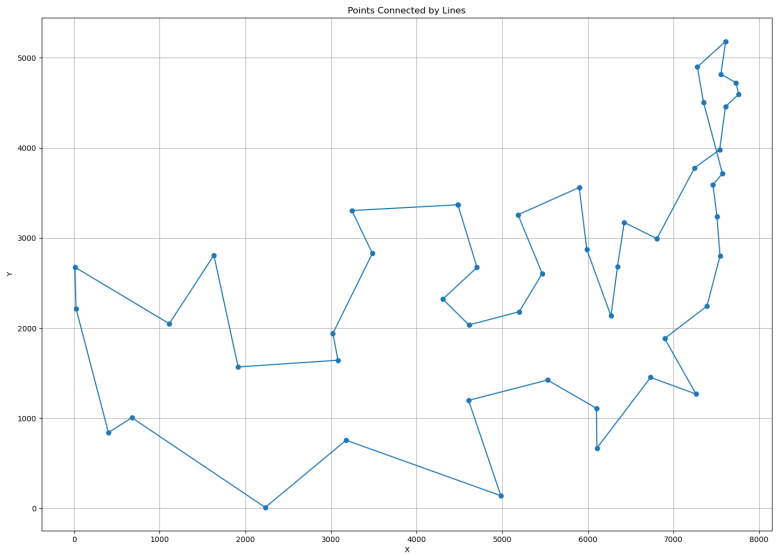

Figure C.1: Best solution on `att48`, 10 minutes with CUDA

- Best result on `kroA100` in 100000 epochs and 64000 individuals

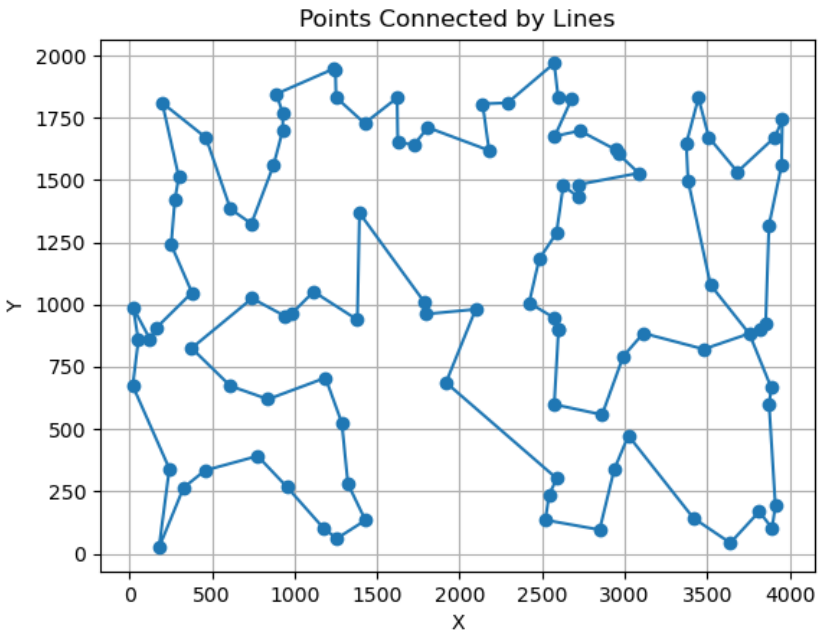

Figure C.2: Good solution on `kroA100`, distance: 22107

- Simple solution of our 11_cities dataset in 250 generations and 32 individuals

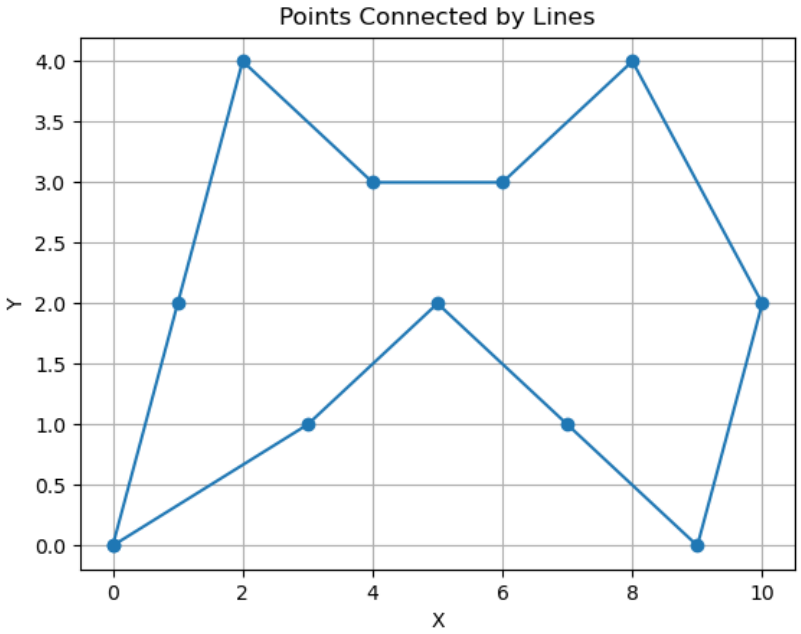

Figure C.3: Optimal solution on `11_cities`, less than 1 second with CUDA

Bibliography

[1] Prasanna Jog, Jung-Yul Suh, and Dirk Van Gucht. The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the traveling salesman problem. 1989.

[2] Traveling salesman problem - cornell university computational optimization open textbook - optimization wiki.

[3] L. Wang, A.A. Maciejewski, H.J. Siegel, and V.P. Roychowdhury. A comparative study of five parallel genetic algorithms using the traveling salesman problem. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 345–349, 1998.

[4] John Runwei Cheng and Mitsuo Gen. Parallel genetic algorithms with gpu computing. In Tamás Bányai and Antonella Petrilloand Fabio De Felice, editors, *Industry 4.0*, chapter 6. IntechOpen, Rijeka, 2020.

[5] Boqun Wang, Hailong Zhang, Jun Nie, Jie Wang, Xinchen Ye, Toktonur Ergesh, Meng Zhang, Jia Li, and Wanqiong Wang. Multipopulation genetic algorithm based on gpu for solving tsp problem. *Mathematical Problems in Engineering*, 2020.

[6] Jean-Yves Potvin. Genetic algorithms for the traveling salesman problem. 2005.