DOI: https://doi.org/10.21203/rs.3.rs-1859168/v1

BrkgaCuda 2.0: A Framework for Fast Biased Random-Key Genetic Algorithms on GPUs

Bruno A. Oliveira[1*], Eduardo C. Xavier[1] and Edson Borin[1]

[1]Institute of Computing, State University of Campinas, Cidade Universitária Zeferino Vaz – Barão Geraldo, Campinas, 13083970, São Paulo, Brazil.

*Corresponding author(s). E-mail(s): b234921@dac.unicamp.br;
Contributing authors: ecx@ic.unicamp.br; edson@ic.unicamp.br;

Abstract

In this paper, we present the development of a new version of the BrkgaCuda, called BrkgaCuda 2.0, to support the design and execution of Biased Random-Key Genetic Algorithms (BRKGA) on CUDA/GPU-enabled computing platforms, employing new techniques to accelerate the execution. We compare the performance of our implementation against the standard CPU implementation called BrkgaAPI, developed by Toso and Resende (2015), and the recently proposed GPU-BRKGA, developed by Alves et al (2021). In the same spirit of the standard implementation, all central aspects of the BRKGA logic are dealt with our framework, and little effort is required to reuse the framework on another problem. The user is also allowed to choose to implement the decoder on the CPU in C++ or on GPU in CUDA. Moreover, the BrkgaCuda provides a decoder that receives a permutation created by sorting the indices of the chromosomes using the genes as keys. To evaluate our framework, we use a total of **54** instances of the Traveling Salesman Problem (TSP), the Set Cover Problem (SCP), and the Capacitated Vehicle Routing Problem (CVRP), using a greedy and an optimal decoder on the CVRP. We show that our framework is faster than the standard BrkgaAPI and the GPU-BRKGA while keeping the same solution quality. Also, when using the *bb-segsort* to create the permutations, our framework achieves even higher speedups when compared to the others.

Keywords: Genetic algorithms, BRKGA, Framework, Parallel, GPU, CUDA

1 Introduction

Several techniques were developed aiming to solve hard combinatorial optimization problems. Among these techniques, there are combinatorial branch-and-bound algorithms, integer linear programming, dynamic programming, and other enumerative methods. Despite the development in this field, some problems with large input instances are still difficult to be tackled with these techniques due to the large amount of time required to explore a large search space. In these cases, the use of heuristics and metaheuristics seems to be appropriate, since one can find acceptable solutions in reasonable computing times. Even when optimal solutions are required, the use of heuristics appears as a standard method to generate primal bounds, helping to speedup the execution of exact algorithms (Sadykov et al, 2019).

Several papers were devoted to the description of heuristics and metaheuristics for hard combinatorial problems, such as Variable Neighborhood

Search, Greedy Randomized Adaptive Search Procedure, Simulated Annealing, and Genetic Algorithms (Ezugwu et al, 2021). This last one belongs to the class of evolutionary algorithms, where a population of individuals, representing solutions, evolve while a selection process occurs filtering the best individuals to the next generations. Among them, the Biased Random-Key Genetic Algorithm (BRKGA) has the advantage of being easy to apply to different types of problems. Some examples of its applications are network design (Andrade et al, 2021; Gonçalves and Resende, 2015), scheduling (Kong et al, 2020; Soares and Carvalho, 2020), molecular docking (Leonhart et al, 2019), routing (Chagas et al, 2021; Ruiz et al, 2019), and set covering (Jing et al, 2020).

To design a solution with BRKGA, the user must specify a decoding function that converts an array of decimal values in the range $[0, 1]$ into solutions. All the other aspects of the algorithm, such as evolution, crossover, and mutation, are standard operations in the BRKGA. Since those operations are independent of the problem, Toso and Resende (2015) proposed a C++ Application Programming Interface (API) that requires only that the user implements the decoding function to the problem. This implementation significantly reduces the effort needed to implement a BRKGA-based solution. Moreover, to speedup the execution time of the algorithm, Xavier (2019) and Alves et al (2021) implemented the same code in CUDA to accelerate the algorithm using Graphics Processing Units (GPU).

The CUDA language was created by Nvidia to allow developers to use the GPU for general processing, not only for graphical processing (Nvidia, 2007). The language provides a layer that gives access to the GPU, which is designed to execute massively parallel tasks. Since its release, several applications were ported to run over CUDA, even heuristics to hard combinatorial optimization problems — in general with significant gains of speed when compared to sequential heuristics (Essaid et al, 2018).

Although metaheuristics do find solutions of good quality, the reuse of the implemented code to tackle different problems requires a significant implementation effort. Some works tried to create libraries to facilitate this task, such as the Jgap (Meffert et al, 2012), which provides a package in Java for genetic algorithm development. Yet, the use of the package is not so simple since the user has to implement several aspects of the algorithm. In this respect, the BrkgaAPI, the BrkgaCuda, and the GPU-BRKGA bring great simplification, requiring only the implementation of the decoding function.

In this work, we present a new version of the BrkgaCuda, called BrkgaCuda 2.0, which improves the original framework proposed by Xavier by employing new techniques to accelerate the execution, such as the use of multiple streams and changing the logic to take advantage of coalesced memory access. Also, we improve the solution quality of the GPU-BRKGA, bringing results similar to the ones found by the BrkgaAPI. It is worth mentioning that the user of our framework is not required to understand GPU programming, since it only requires the implementation of a decoding function that can be implemented in standard C++ language — in this case, our framework is still able to harness the power of GPUs to accelerate the BRKGA. To evaluate our framework, we used 54 instances, 22 instances of the Traveling Salesman Problem (TSP) and 9 instances of the Set Cover Problem (SCP) with instances available in the OR-Library[1], and 23 instances of the Capacitated Vehicle Routing Problem (CVRP) with instances available in the CVRPLIB[2]. We perform an extensive evaluation by comparing its performance against the BrkgaAPI, the BrkgaCuda, and the GPU-BRKGA using four implementations to these problems. We show that the BrkgaCuda 2.0 is the fastest one, create solutions of similar quality, and doesn't have the chromosome length limited by the number of CUDA threads.

The remainder of this text is organized as follows: Section 2 provides an overview of Biased Random-Key Genetic Algorithms; Section 3 presents the related works; Section 4 discusses the implementation of the BrkgaCuda 2.0; Section 5 presents several computational experiments comparing the BrkgaCuda 2.0 with the BrkgaAPI, the BrkgaCuda, and the GPU-BRKGA; finally, Section 6 presents the conclusions.

[1] http://people.brunel.ac.uk/~mastjjb/jeb/info.html
[2] http://vrp.atd-lab.inf.puc-rio.br/index.php/en/

2 Biased Random-Key Genetic Algorithm

The Genetic Algorithm (Holland, 1984) was developed as a method for solving optimization problems simulating the natural selection process. The algorithm simulates a population of individuals, where each individual is represented by a chromosome, which in turn represents a solution. Each chromosome consists of n genes. The evolution of the population combines the genes of individuals, a process called crossover. Although the process is random, the crossover process is specified in such a way that good characteristics of the chromosomes are passed to the next generation. The individuals can also mutate, helping to escape from local optima. This mutation can be done, for example, by creating new individuals with random genes or selecting some individuals after the crossover operation and changing some genes to a new random value. Also, the concept of survival is applied as a way of selecting individuals representing the best solutions for the next generation.

In the BRKGA, the genes of the chromosome are represented by a random key encoded as a floating-point value in the range $[0, 1]$. Initially, a population is randomly created with p chromosomes. Then the fitness is calculated by the decoding function, which maps the chromosome to a numerical value (the fitness) that represents the solution performance (or quality). After calculating the fitness of each chromosome, the population is divided into two groups: the p_e best solutions (elite) and the $p - p_e$ others (non-elite). The population is evolved through survival, mutation, and crossover operations, which change the population across generations. The survival operation ensures that the p_e best chromosomes of the current generation are preserved in the next generation. The mutation operation constructs p_m new chromosomes with random values. Finally, the remaining $p - p_e - p_m$ chromosomes are generated by the crossover operation, which combines one chromosome from the elite group with other from the non-elite groups. Let $\rho > 0.5$ be a parameter for the crossover operation, the new chromosome C is created by combining an elite (C^e) and a non-elite (C^{ne}) chromosome so that gene $C_i = C_i^e$ with probability ρ and $C_i = C_i^{ne}$ with probability $1 - \rho$, for each $i = 1, \ldots, n$. Note that ρ is biased to select the genes of the elite solution. Figure 1 presents the BRKGA flowchart, highlighting the only part that is problem-dependent.

Finally, the multi-population concept increase the variability of good solutions in each population. In this concept, several populations are created and evolve independently of each other. To find even better solutions, it is possible to perform periodic exchanges of the best individuals between different populations, replacing bad chromosomes with the elites of another population (Branke et al, 2000).

3 Related Works

Several works proposed Biased Random-Key Genetic Algorithms to solve optimization problems. Chagas et al (2021) combine the TSP with the knapsack problem to select the locations to visit that maximize the profit while minimizing the travel time. To do so, they modified the BRKGA to work in a Multi-Objective fashion and concluded that their solution consistently finds high-quality solutions. Ruiz et al (2019) apply the BRKGA to the Open Vehicle Routing Problem (OVRP), which doesn't require the vehicle to return to the depot at the end of the tour. Their solution could improve the results of 16 out of 30 instances tested. Cicek and Ozturk (2021) use the BRKGA to define the number of neurons and their weights of an Artificial Neural Network (ANN). This approach could find better forecasts when compared to other ANN algorithms.

BRKGA has also been combined with other techniques to improve its convergence and the solution quality. Chaves et al (2018) proposed to make the parameters of the BRKGA adaptive, *i.e.*, updating the parameters while the algorithm is running. This approach simplified even more the algorithm by removing the need to set the parameters of the algorithm while improving the results compared to static parameters. Jing et al (2020) combine the SCP and the VRP to solve the problem of using drones to perform inspections. They use the BRKGA with local search to solve the problem, reducing the total distance of the planned inspection to near half of the ones found in the literature. Andrade et al (2021) add to the BRKGA the Multi-Parent strategy, which selects three or more chromosomes to mate to generate the next population. Moreover, they applied

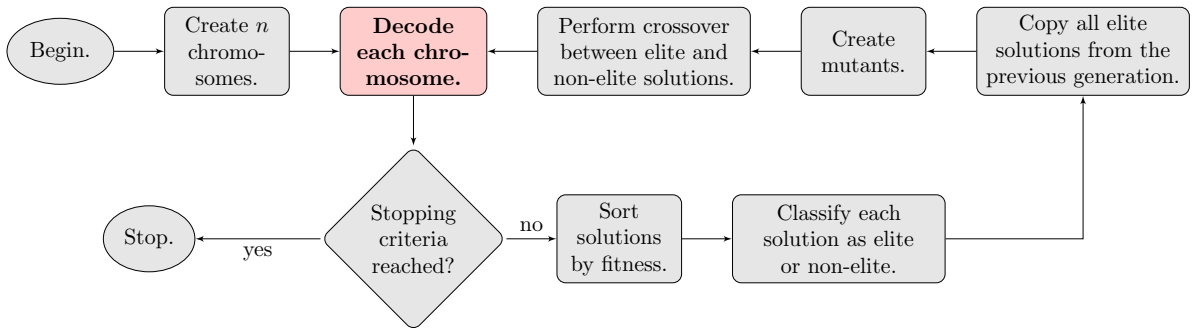

Fig. 1 Flowchart of the BRKGA algorithm.

the Implicit Path Relinking (IPR) local search independently of the problem, allowing its reuse on other problems. Soares and Carvalho (2020) address the scheduling problem using the BRKGA with the Variable Neighborhood Descend running many local searches. Homayouni et al (2020) address a variation of the Job-Shop Scheduling Problem (JSP) using the BRKGA with many heuristics performing local searches. All works that used BRKGA with local search had faster convergence and found solutions of very high quality.

Some frameworks were proposed to facilitate the design and implementation of BRKGA-based solutions. These frameworks aim at abstracting away the details, leaving the user only with the design and implementation of a solution decoding function, which takes a chromosome as input and returns a numerical value (the fitness) indicating the performance (or quality) of the solution it represents. Smaller fitness means better chromosomes.

A C++ implementation known as BrkgaAPI was proposed by Toso and Resende (2015). The BrkgaAPI provides the user with the `BRKGA` class, which is implemented as a template class that receives a `Decoder` class as a parameter. The `Decoder` class, in turn, is a C++ code defined by the user that implements the decoder function. Xavier (2019) presented the first version of the BrkgaCuda, a framework that implements a similar API, but runs on GPUs. This framework allows the user to implement the decoding process for execution on the CPU or the GPU and achieved performances up to 1.36× faster than BrkgaAPI using OpenMP. A similar framework called GPU-BRKGA was proposed by Alves et al (2021). The authors evaluated the GPU-BRKGA performance using three multidimensional functions and concluded that it is faster than the BrkgaAPI and the BrkgaCuda frameworks. It is worth noticing that these problems are not representative of combinatorial optimization problems, and they were evaluated using small chromosomes (≤ 128). As we show in our experimental results, their implementation does not work with large chromosomes (> 1024) and did not perform as well as our implementation on hard combinatorial optimization problems, which are more common on the literature.

In this work, we present a new version of the BrkgaCuda, called BrkgaCuda 2.0, which improves the preliminary framework proposed by Xavier by employing new techniques to accelerate the execution. Finally, we perform an extensive evaluation by comparing its performance against the BrkgaAPI, the previous version of the BrkgaCuda, and the GPU-BRKGA frameworks using 54 instances with four implementations of three combinatorial optimization problems.

4 The BrkgaCuda 2.0 Framework

In this work, we present the BrkgaCuda 2.0, an open-source framework[3] to facilitate the design and execution of fast Biased Random-Key Genetic Algorithms on GPUs. In this section, we present the framework, a sample decoder for the TSP problem, and the parallelization of the BRKGA using CUDA.

[3]To be made available at `GitHub`.

4.1 The BrkgaCuda 2.0

The main class in this framework is the `Brkga` class, which contains the following public methods:

- `Brkga`: a constructor method that takes as a parameter a configuration class containing: the decoder of the problem; the number of independent populations; the population size; the size of the chromosome; the number or the percentage of elites; the number or the percentage of mutants; the bias used in the crossover operation; the decoder to use; the seed used in the pseudo-random number generator; the number of threads to use on the CPU and the GPU.
- `evolve`: evolves the populations to the next generation.
- `exchangeElite(M)`: exchanges the best M elites of each population, replacing the worst chromosomes.
- `getBestFitness`, `getBestChromosome`, and `getBestPermutation`: return, respectively, the fitness, the chromosome, and the permutation of the best fitness found among all populations.

The other relevant class is the abstract class `Decoder`, which contains the methods that must be implemented by the user. The user should override the method according to the desired decoder. A common operation is to sort the chromosome before decoding, especially when the solution is a permutation-like. The "permutation" methods already provide the user with the permutation created by sorting the genes of the chromosome and returning their initial index. In this case, the sorting process is performed on the GPU by the framework itself. The available decoders are:

- Single-CPU: receives a single chromosome and must return its fitness value.
- Single-CPU-Permutation: similar to Single-CPU, but receives a permutation instead of the chromosome.
- CPU: receives a population of chromosomes and an array in which the user must store the results. The code must calculate and store the fitness value of each chromosome on the results array.
- CPU-Permutation: similar to CPU, but receives a permutation instead of the chromosome.
- GPU: similar to CPU, but runs on the GPU.
- GPU-Permutation: similar to CPU-Permutation, but runs on the GPU.

All decoders that run on the GPU receive a stream to run on. The user is also allowed to specify that all populations must be decoded at once, allowing its implementation to process a large amount of data at once and, perhaps, have a speedup due to better resource usage. Moreover, the default implementation of CPU and CPU-Permutation is to call, respectively, Single-CPU and Single-CPU-Permutation using OpenMP, simplifying the implementation. The GPU and GPU-Permutation were provided for users who have knowledge of CUDA and want to use the GPU to decode the chromosome. Otherwise, the user can implement the CPU decoder in plain C++. Notice that, even though the user may implement all methods, only one of them needs to be implemented to produce a valid BRKGA solution. The next section illustrates how the framework can be used through the implementation of a sample decoder for the TSP problem.

4.2 Sample Decoder

Consider for example the TSP problem. The input is a complete graph $G = (V, E)$ with edge costs and whose goal is to find a minimum cost Hamiltonian cycle. We can consider the chromosomes with size $n = \mathsf{V}$, in which each gene represents a vertex. Then we sort the genes according to their key values keeping track of the initial index. This way the indices will generate a permutation representing an order to visit the vertices in a Hamiltonian cycle. The fitness will be the sum of the costs of the edges used to visit the clients in that order.

The Code 1 presents an example of the decoder discussed above for the BrkgaCuda 2.0. In this example, the clients are represented by 2D coordinates and the graph G is implicit with edge costs calculated by the `distance` method. The decoder inherits from `Decoder` and overrides the `decode` method for the chromosome and a population of permutations, both on the CPU. The GPU version is similar.

```
class TspDecoder : public box::Decoder {
std::vector<Point> clients;  // 2D points

public:
using box::Decoder;

TspDecoder(std::vector<Point> c)
    : clients(c) {}

float distance(unsigned u,
               unsigned v) const {
  return std::hypotf(
      clients[u].x - clients[v].x,
      clients[u].y - clients[v].y);
}

// Decode the chromosome on the CPU
float decode(const float* chromosome)
    const override {
  // config is defined in box::Decoder
  unsigned n = config->chromosomeLength;

  // Sort the permutation using
  //   the chromosome as keys
  std::vector<unsigned> tour(n);
  std::iota(tour.begin(), tour.end(), 0);
  std::sort(tour.begin(), tour.end(),
    [&](unsigned lhs, unsigned rhs) {
      return chromosome[lhs]
           < chromosome[rhs];
    });

  // Calculate the fitness
  float fitness = distance(tour[0],
                           tour[n - 1]);
  for (unsigned j = 1; j < n; ++j) {
    fitness += distance(tour[j - 1],
                        tour[j]);
  }
  return fitness;
}
};
```

Code 1 Sample decoder of the TSP for the BrkgaCuda 2.0.

4.3 Parallelization of the BRKGA on CUDA

To represent all the chromosomes in all populations we use a contiguous array. We use the `curandGenerateUniform` function, from the CUDA toolkit, to generate pseudo-random numbers in the range $[0, 1]$ in parallel. This generator is used to initialize and evolve the population. The evolution requires a temporary array to which we copy elite chromosomes, create the mutants, and perform the crossover. To create the elite and non-elite groups, we keep a sorted array of fitness and the index of the corresponding chromosome. The first p_e chromosomes belong to the elite group and the last p_m are the ones replaced by the mutants.

In the new version of the framework, each thread processes a single gene. To sort all chromosomes at once (if the user chooses to decode the permutations), we use the *bb-segsort* implementation by Hou et al (2017), as suggested by Schmid et al (2022). Moreover, each population is assigned to a different stream. The multiple streams allow scheduling CUDA operations to run concurrently, enabling the GPU to interleave the execution with data transfers and functions that doesn't use all the resources of the GPU, for example. This way, the operations on the same population are enqueued and the synchronization only happens in three cases: to call the *bb-segsort*, to exchange elites, and to return the best chromosome. Moreover, the synchronization to call the *bb-segsort* occurs only if the user selects to use the sorted decoder. As we show in the experiments, even synchronizing before calling the sorted decoder have higher speedups.

Figure 2 presents the expected CPU and GPU usage executing the BrkgaCuda 2.0 when evolving three populations for four generations, exchanging the elites after the 2nd generation. Each population is assigned to a different stream (P_1, P_2, and P_3). The filled boxes represent a kernel launch, which enqueues the operation to be processed on a stream. On the beginning, the evolution (ev) is enqueued for each population. The evolution doesn't occur concurrently as one population will already be using all the GPU resources. As the CUDA blocks finishes, they will release some resources, allowing another enqueued evolution to start running concurrently. Notice that the CPU is blocked waiting the evolution to finish before starting to decode (dec). When the decoding is done, the corresponding population is sorted (S) to define the elite and non-elite groups, as well as defining the best chromosome of the population. On this point, the GPU gets blocked waiting the decoder, and the CPU cannot enqueue the evolve operation because it is waiting the call to the evolve method return. This process will repeat to all generations, which after some of them will execute the exchange elites (X) method. In the example, the method is called after the 2$^{\text{nd}}$ generation. This method depends on all populations, and the next generation also depends on it, blocking the CPU and the GPU until the exchange finishes.

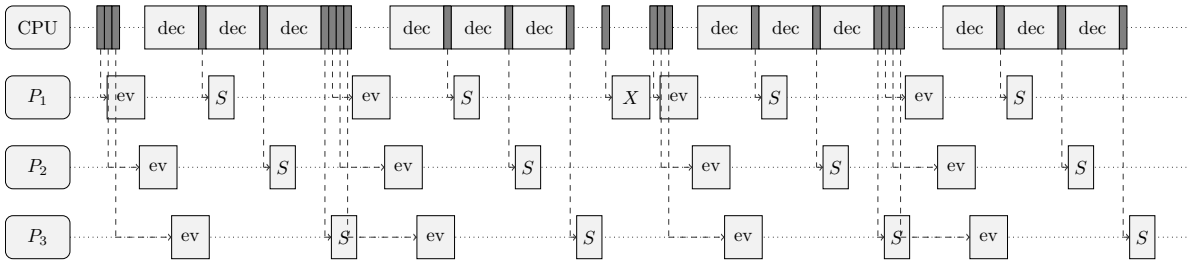

Fig. 2 Expected CPU and GPU usage when running 4 generations of 3 populations decoding on the CPU, exchanging elites after every 2nd generation.

5 Computational Experiments

In this section, we present computational results to compare the performance of the BrkgaCuda 2.0 against the BrkgaAPI, BrkgaCuda, and the GPU-BRKGA. The BrkgaAPI can only decode on the CPU while the GPU-BRKGA uses CUDA and can execute the decode operation on the CPU or the GPU. In addition, the BrkgaCuda can also decode the permutation on the GPU. Our framework also uses CUDA and can decode on the CPU or the GPU; nonetheless, it also allows the user to decode a permutation based on the genes' indices sorted according to their key value.

The machine used in the experiments has an Intel(R) Core(TM) i7-7820X[4] with 64GiB of memory and a Nvidia GeForce GTX 1070 GPU with 8GiB of memory. We ran the experiments inside a Docker container with *nvcc* 11.2.152, and *g++* 9.4.0 . The experiments consist of searching for solutions to TSP, SCP, and CVRP using the BrkgaAPI, the GPU-BRKGA, the BrkgaCuda, and the BrkgaCuda 2.0 frameworks. The tests evaluate the algorithm alone, *i.e.*, without any local search.

In the tests, we evaluate all decoders available in each framework. Due to the stochastic nature of the BRKGA, each test is repeated 10 times. We present on Figure 3, for each problem, a boxplot with the quality of the solutions obtained normalized using the average of the values found by the BrkgaAPI on each instance. We choose to normalize using the BrkgaAPI because it is the first and standard framework used. Each box of the plot is divided into four sorted ranges, highlighting the lowest, the highest, the median, the median of the lower half, and the median of the upper half of the values. Outliers are plotted as individual points. Let's take as an example the results with the GPU-BRKGA (highlighted). The figure shows that, except for the TSP, both of its decoders (CPU and GPU) has more than 50% of the values worse than the ones found with the BrkgaAPI, and near to 75% are worse than most of them. Moreover, the smallest solution found with the GPU-BRKGA is close to the median found with the BrkgaAPI. It was expected that all frameworks keep the solution close to the same range. This consistently worse solutions motivated us to investigate the implementation and fix its sorting procedure to work on all populations during the evolution, calling that version GPU-BRKGA-Fixed. This new version have similar values to the ones found with BrkgaAPI; in fact, it was able to find even better results on the SCP. Also, as our experimental results show, this new version achieves the same performance as GPU-BRKGA.

The parameters chosen for the frameworks are: $k = 3$ (populations), $p = 256$ (chromosomes per population), $p_e = 25$ (elites), $p_m = 25$ (mutants), and $\rho = 0.75$ (bias). The algorithm runs for 1000 generations, exchanging the 2 best chromosomes after every 50 generations. To define the parameters we selected a small set of instances to perform the tuning. We ran all frameworks to ensure that our algorithm doesn't have an unfair advantage over the others. The results (fitness ratio and speedup) are similar with different parameters, except for GPU-BRKGA, which doesn't improve the fitness when increasing the number of populations.

5.1 Results with TSP

Given $G = (V, E)$ with edge costs, the TSP consists of determining a route to visit $n = \mathsf{V}$ cities (the nodes of G) exactly once, returning to

[4]This CPU contains a total of 8 cores and 16 hardware threads.

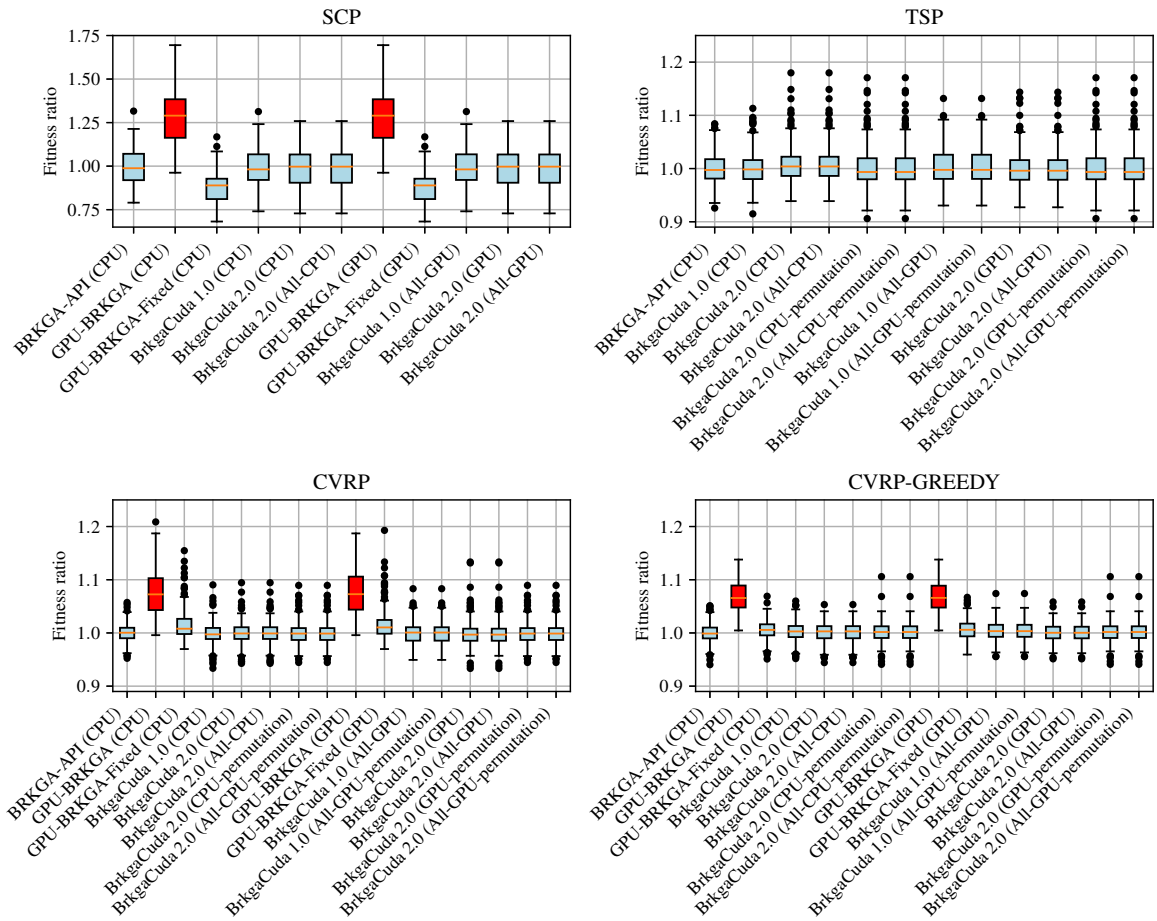

Fig. 3 Range of the fitness ratio obtained with each implementation on the SCP, the TSP, and the CVRP.

the origin city at the end of the tour. The objective is to find a route with the minimum cost (such as distance or time). In this problem, we set the length of the chromosome to n. To decode, we use a permutation as explained in Section 4.2. This strategy was used both on the CPU and on the GPU implementation.

Figure 4 presents the boxplot of the time elapsed of each framework for the TSP. The GPU-BRKGA failed to execute due to the chromosome length exceeding the maximum number of threads of the GPU (which is 1024); hence, we decided to not include its results. When decoding the chromosome on the CPU, both versions of the BrkgaCuda have speedups between 2× and 4×, while decoding the permutation have speedups between 4× and 7×. When decoding the chromosome on the GPU, the speedups are between 0.5× and 3×. Decoding the permutations on the GPU have even higher speedups of up to 10×. Note that, when decoding on the CPU, the decoder will use all the CPU cores to decode in parallel. Investigating the speedup obtained when decoding on the GPU, we found that the libraries to sort the chromosome (both thrust and *bb-segsort*) used by the decoder performs a synchronization. Moreover, in the BrkgaCuda 1.0, all populations are decoded at once, using all the GPU resources. Decoding all chromosomes on a single call to the decoder will allow the user's implementation to harness the GPU resources and reduce the number of synchronizations. Compared to the BrkgaCuda 2.0 GPU decoder, the All-GPU decoder of the BrkgaCuda 1.0 had a speedup on the median time elapsed of 1.4× and the BrkgaCuda 2.0 had a speedup of 1.6×. The All-GPU-Permutation of the BrkgaCuda 2.0 had a speedup of 1.5× of the

median value when compared to the GPU-Permutation.

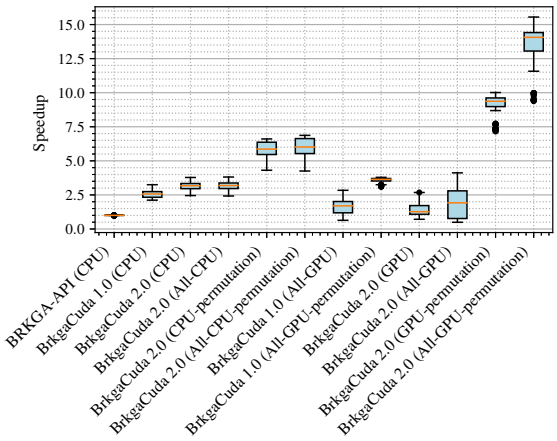

Fig. 4 Boxplot of the time elapsed of each framework for the TSP.

5.2 Results with SCP

Given a universe U and n sets $S_i \subset U, i = 1, \ldots, n$, each set with a cost associated, the SCP consists of selecting some of those sets whose union equals U and the cost is minimized. In other words, select some sets in a way that each element of U is covered by at least one set. In this problem the length of the chromosome is n. To decode the chromosome, the set i in the solution if gene $C_i < 0.5$. If an element $u \in U$ is uncovered, the fitness of the chromosome is set to infinity. Moreover, this decoder doesn't require sorting the chromosome. Thus, we didn't run the experiments with the decoders of the permutation.

Figure 5 presents the boxplot of the time elapsed of each framework for the SCP. Most of the decoders achieved speedups between 2× and 3×. As in the TSP, decoding all chromosomes at once allow the decoder to harness the GPU resources. The `All-GPU` decoder has a speedup near to 6× and 7.5× with BrkgaCuda and BrkgaCuda 2.0, respectively.

5.3 Results with CVRP

Given a complete graph $G = (V, E)$, the CVRP consists of determining many routes starting on a depot to deliver the demands of $n = \mathsf{V} - 1$ clients

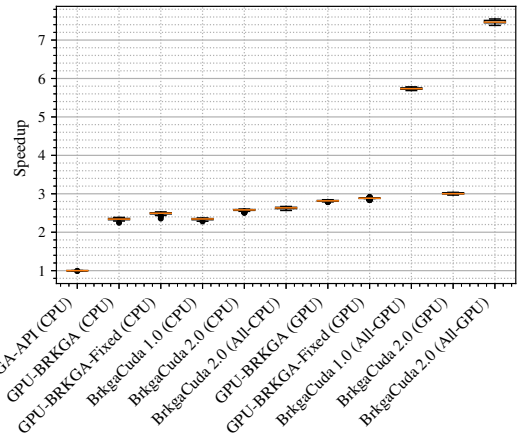

Fig. 5 Boxplot of the time elapsed of each framework for the SCP.

minimizing the cost, visiting each client exactly once. The vehicles have the same capacity and each demand has weight d_i. In this problem, the chromosome has length n, and to decode we use a permutation as explained in Section 4.2. To split the permutation into routes we use dynamic programming to find in linear time the optimal clients to return to the depot, as explained by Vidal (2016). The solution will be the cost between adjacent clients plus the cost from the depot to the clients and from the clients back to the depot on the split positions.

Figure 6 presents the boxplot of the time elapsed of each framework for the CVRP when using dynamic programming. To decode on the device we use the same dynamic programming, which isn't good to use on the GPU due to the large number of memory accesses that cannot be coalesced. Moreover, the decoder will synchronize when decoding the chromosomes as we discussed. Thus, the GPU decoders don't perform as good as the CPU decoders; in fact, the GPU decoder is slower than the BrkgaAPI which runs on the CPU only. Even the fastest decoder (All-GPU-Permutation) could not surpass the CPU decoder of the GPU-BRKGA, the GPU-BRKGA-Fixed, and the BrkgaCuda 2.0.

Since the dynamic programming algorithm did not perform well on the GPU, we tested a greedy decoder to evaluate time elapsed. The greedy strategy consists in picking the clients from the permutation, one by one, until the truck is full and another route is started. Remember that the

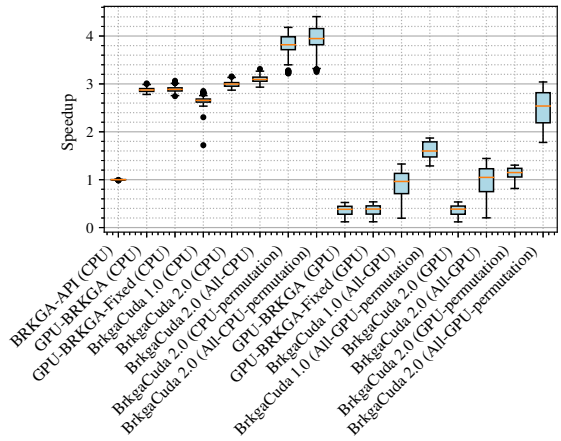

Fig. 6 Boxplot of the time elapsed of each framework for the CVRP with dynamic programming.

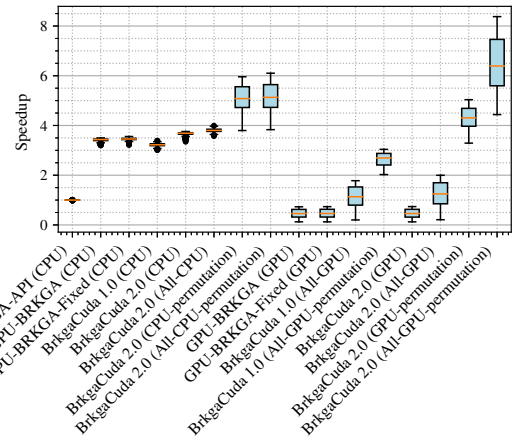

Fig. 7 Boxplot of the time elapsed of each framework for the CVRP with a greedy strategy.

decoder is developed by the user of the framework, and this greedy strategy is fast but generates worse solutions — even with the chromosome corresponding to the best tour this decoder can return a significantly worse fitness value. The fitness with this decoder are on average 30% and up to 65% worse when compared to the dynamic programming decoder. Also, it is important to highlight that the time elapsed with the BrkgaAPI is slightly faster than the ones using dynamic programming.

Figure 7 presents the boxplot of the time elapsed of each framework for the CVRP with the greedy strategy described. Unfortunatelly, the speedups are similar to the ones found with the dynamic programming approach, with a slightly higher speedup.

6 Conclusions

In this paper, we present a new version of the framework BrkgaCuda, called BrkgaCuda 2.0 for the BRKGA using the CUDA platform. BRKGA is a metaheuristic based on genetic algorithms and the use of a generic implementation to it has a great advantage for users since the work of implementing a solution to an optimization problem is significantly reduced. The user has just to implement a decoding function that transforms a random key array (called chromosome) into a solution to the problem being considered. Our framework allows the user to select to use the CPU or the GPU to execute the decoding procedure, and also allows selecting to decode the chromosome or the permutation of the indices of the chromosome when sorting them using the genes as keys.

We compared the BrkgaCuda 2.0 with the standard framework for BRKGA, called BrkgaAPI, the preliminary BrkgaCuda for GPUs, and with the recently proposed GPU-BRKGA also for GPUs. We use the Traveling Salesman Problem (TSP), the Set Covering Problem (SCP), and the Capacitated Vehicle Routing Problem (CVRP) to perform the experiments. Most of the solutions found are in a range of 5% around the average found by the BrkgaAPI, except the GPU-BRKGA that consistently found results worse than BrkgaAPI. That motivated us to investigate the code and fix a sorting procedure that wasn't called on all populations, which we call GPU-BRKGA-Fixed. The fixed version found solutions in the range of 5% as the other frameworks. Thus, we can conclude that both the GPU-BRKGA-Fixed and the BrkgaCuda 2.0 could keep the same quality as the BrkgaAPI. Moreover, the fix didn't have a big impact on the time elapsed when compared to the GPU-BRKGA — it is a bit faster, as we show in the results.

As we presented, even when decoding the chromosomes on the CPU using the `Host` decoder one can expect an improved execution time. In our experiments, the BrkgaCuda 2.0, the GPU-BRKGA, and the GPU-BRKGA-Fixed frameworks required less than half of the time elapsed by

the BrkgaAPI to achieve a similar solution. Moreover, decoding the permutation with the `Host Sorted` decoder was even faster. We also show that decoding on the GPU using the `Device` decoder can have lower speedups if the decoding process does not fit well the GPU parallelism — and, perhaps, have a slowdown. From the problems we have chosen to perform the experiments, only the TSP decoder had higher speedups in the GPU when compared to the CPU, and the CVRP had a slowdown. That is expected because both the SCP and the CVRP decoder require a higher number of memory access that cannot be coalesced. Still, except for the CVRP with dynamic programming, the `Device Sorted` decoder was able to improve the time elapsed when decoding on the GPU.

The BrkgaCuda 2.0 framework does find acceptable solutions in a reasonable time, but in the literature, it is common to find that metaheuristics are combined with local search to further improve a solution by searching on its neighborhood. Thus, the framework can be expanded to provide such local searches to the user and enable them if he wishes to. Moreover, it is interesting to allow the user to control the details of the algorithm, such as adaptive parameters and hybridization with other methods.

References

Alves D, Oliveira D, Andrade E, et al (2021) GPU-BRKGA: A GPU accelerated library for optimization using the biased random-key genetic algorithm. IEEE Latin America Transactions 20(1):14–21. https://doi.org/10.1109/TLA.2022.9662169

Andrade CE, Toso RF, Gonçalves JF, et al (2021) The multi-parent biased random-key genetic algorithm with implicit path-relinking and its real-world applications. European Journal of Operational Research 289(1):17–30. https://doi.org/10.1016/j.ejor.2019.11.037

Branke J, Kaußler T, Smidt C, et al (2000) A multi-population approach to dynamic optimization problems. In: Parmee IC (ed) Evolutionary Design and Manufacture. Springer, London, pp 299–307, https://doi.org/10.1007/978-1-4471-0519-0_24

Chagas JBC, Blank J, Wagner M, et al (2021) A non-dominated sorting based customized random-key genetic algorithm for the bi-objective traveling thief problem. Journal of Heuristics 27(3):267–301. https://doi.org/10.1007/s10732-020-09457-7

Chaves AA, Gonçalves JF, Lorena LAN (2018) Adaptive biased random-key genetic algorithm with local search for the capacitated centered clustering problem. Computers and Industrial Engineering 124:331–346. https://doi.org/10.1016/j.cie.2018.07.031

Cicek ZIE, Ozturk ZK (2021) Optimizing the artificial neural network parameters using a biased random key genetic algorithm for time series forecasting. Applied Soft Computing 102:107,091. https://doi.org/10.1016/j.asoc.2021.107091

Essaid M, Idoumghar L, Lepagnot J, et al (2018) GPU parallelization strategies for metaheuristics: a survey. International Journal of Parallel, Emergent and Distributed Systems 34:1–26. https://doi.org/10.1080/17445760.2018.1428969

Ezugwu AE, Shukla AK, Nath R, et al (2021) Metaheuristics: a comprehensive overview and classification along with bibliometric analysis. Artificial Intelligence Review 54(6):4237–4316. https://doi.org/10.1007/s10462-020-09952-0

Gonçalves JF, Resende MGC (2015) A biased random-key genetic algorithm for the unequal area facility layout problem. European Journal of Operational Research 246(1):86–107. https://doi.org/10.1016/j.ejor.2015.04.029

Holland JH (1984) Genetic Algorithms and Adaptation, Springer US, Boston, MA, pp 317–333. https://doi.org/10.1007/978-1-4684-8941-5_21

Homayouni M, Fontes D, Gonçalves J (2020) A multistart biased random key genetic algorithm for the flexible job shop scheduling problem with transportation. International Transactions in Operational Research pp 1–29. https://doi.org/10.1111/itor.12878

Hou K, Liu W, Wang H, et al (2017) Fast segmented sort on GPUs. In: Proceedings of the International Conference on Supercomputing. Association for Computing Machinery, Chicago, USA, https://doi.org/10.1145/3079079.3079105

Jing W, Deng D, Wu Y, et al (2020) Multi-UAV coverage path planning for the inspection of large and complex structures. pp 1480–1486, https://doi.org/10.1109/IROS45743.2020.9341089

Kong M, Liu X, Pei J, et al (2020) A BRKGA-DE algorithm for parallel-batching scheduling with deterioration and learning effects on parallel machines under preventive maintenance consideration. Annals of Mathematics and Artificial Intelligence 88(1):237–267. https://doi.org/10.1007/s10472-018-9602-1

Leonhart PF, Spieler E, Ligabue-Braun R, et al (2019) A biased random key genetic algorithm for the protein–ligand docking problem. Soft Computing 23(12):4155–4176. https://doi.org/10.1007/s00500-018-3065-5

Meffert K, Rotstan N, Knowles C, et al (2012) JGAP: Java genetic algorithms and genetic programming package. http://jgap.sourceforge.net/

Nvidia (2007) CUDA: Compute unified device architecture programming guide. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, visited in: 11-09-2019

Ruiz E, Soto-Mendoza V, Ruiz Barbosa AE, et al (2019) Solving the open vehicle routing problem with capacity and distance constraints with a biased random key genetic algorithm. Computers & Industrial Engineering 133:207–219. https://doi.org/10.1016/j.cie.2019.05.002

Sadykov R, Vanderbeck F, Pessoa A, et al (2019) Primal heuristics for branch and price: The assets of diving methods. INFORMS Journal on Computing 31(2):251–267. https://doi.org/10.1287/ijoc.2018.0822

Schmid RF, Pisani F, Cáceres EN, et al (2022) An evaluation of fast segmented sorting implementations on GPUs. p 102889, https://doi.org/10.1016/j.parco.2021.102889

Soares LCR, Carvalho MAM (2020) Biased random-key genetic algorithm for scheduling identical parallel machines with tooling constraints. European Journal of Operational Research 285(3):955–964. https://doi.org/10.1016/j.ejor.2020.02.047

Toso RF, Resende MGC (2015) A C++ application programming interface for biased random-key genetic algorithms. Optimization Methods and Software 30(1):81–93. https://doi.org/10.1080/10556788.2014.890197

Vidal T (2016) Technical note: Split algorithm in O(n) for the capacitated vehicle routing problem. Computers & Operations Research 69:40–47. https://doi.org/10.1016/j.cor.2015.11.012

Xavier EC (2019) Uma interface de programação de aplicações para o BRKGA na plataforma CUDA. In: XX Simpósio de Sistemas Computacionais de Alto Desempenho. SBC, Porto Alegre, RS, Brasil, pp 13–24, https://doi.org/10.5753/wscad.2019.8653

Statements and Declarations

Funding Bruno Almêda de Oliveira has received research support from Santander Universidades. Edson Borin received research support from CNPq (314645/2020-9 and 404087/2021-3) and from CCES/FAPESP (2013/08293-7). Authors would also like to thank the Multidisciplinary High Performance Computing Lab (LMCAD-IC) for its infrastructure and technical support.

Competing Interests The authors have no relevant financial or non-financial interests to disclose.

Author Contributions Conceptualization was performed by Eduardo Candido Xavier. Implementation was performed by Bruno Almêda de Oliveira and Eduardo Candido Xavier. Experimental design and results analysis were performed by Bruno Almêda de Oliveira and Edson Borin. The manuscript was mainly written by Bruno

Almêda de Oliveira and extensively reviewed by all authors.

Data Availability The datasets generated and analysed during the current study are available at the GitHub, [to be made available].