sciendo
Control and Cybernetics
vol. **53** (2024) No. 2
pages: 371-383
DOI: 10.2478/candc-2024-0016

Accelerating the Clarke-Wright algorithm using GPUs*

by

Francesca Guerriero and Francesco Paolo Saccomanno

Department of Mechanical, Energy and Management Engineering,
University of Calabria, Italy
francesca.guerriero@unical.it,
francescopaolo.saccomanno@unical.it

Abstract: The Capacitated Vehicle Routing Problem (CVRP) is a combinatorial optimization problem that seeks to determine the optimal set of routes for a fleet of vehicles, with limited capacity, to deliver goods to customers while minimizing the total cost. Due to its NP-hard nature, finding exact solutions for the large-scale CVRP instances is computationally intractable. Therefore, heuristics and metaheuristics are widely employed to find approximate optimal solutions. Among these, the Clarke-Wright (CW) algorithm is a popular greedy approach that constructs routes by iteratively merging nodes to minimize transportation costs. This study presents an implementation of the CW algorithm in graphics processing units (GPUs) using the CUDA (Compute Unified Device Architecture) framework. The GPU implementation is compared to its CPU counterpart in terms of execution time and performance. The results demonstrate significant speed-ups achieved by the GPU implementation, particularly for large-scale instances. Performance gains can be attributed to the parallel processing capabilities of GPUs, enabling efficient execution of the algorithm computational steps.

Keywords: capacitated vehicle routing problem, CVRP, Clarke-Wright Algorithm, GPU Computing, CUDA, heuristics

1. Introduction

The goal of this study is to explore the implementation of the CW algorithm (Clarke and Wright, 1964), to solve the CVRP on modern GPU platforms, to analyse its execution times by breaking down its course according to the various steps of the algorithm, and to compare them with their CPU-based counterpart.

*Paper presented at the BOS/SOR 2024 conference in October 2024. Accepted in November 2024.

The CVRP is a challenging optimization problem, whose objective is to determine the optimal set of routes for a certain number of vehicles, of fixed capacity, to deliver products to customers, while minimizing the total transportation cost. Being a combinatorial optimization problem of the NP-Hard type (Laporte, 1992), it can be solved exactly only for a limited number of nodes (i.e. deliveries to be made) and vehicles. In cases other than the simplest ones, heuristic and metaheuristic approaches have been developed to find good-quality solutions in a reasonable amount of time (Accorsi and Vigo, 2021; Liu et al., 2023). Among these, a widely used greedy approach is the CW savings algorithm (Augerat et al., 1995). It is known for its simplicity, efficiency, and effectiveness in producing high-quality solutions.

The CW algorithm relies on building a list of possible savings, followed by iterations that swap visited nodes. This greedy approach, whose operations will be explained in detail in the following sections, requires the calculation of cost matrices and savings lists between nodes, whose processing can be optimized using a parallel approach, in particular, the one based on CUDA.

CUDA is a parallel computing platform and application programming interface (API) developed by NVIDIA in 2007, Hijma et al. (2023). It enables programmers to utilize the computational power of Graphics Processing Units (GPUs) for general-purpose computing tasks.

CUDA employs a single-instruction multiple-thread (SIMT) architecture. In this paradigm, the GPU serves as a coprocessor for the CPU. The CPU manages the overall code execution, while tasks that require parallel processing are delegated to the GPU. These parallel tasks are encapsulated in units called kernels. Each kernel is executed by a grid, which is logically divided into blocks. Each block can execute a specific number of threads, currently up to 1024.

Modern CPUs and GPUs have different architectures that make them suitable for different types of tasks. CPUs have a few very complex processing units, which are capable of executing a wide range of instructions. They are optimized for tasks that require high precision and low latency, such as running operating systems, web browsers, and other applications.

On the other hand, GPUs have hundreds of very simple processing units, which are capable of executing a large number of instructions in parallel. They are optimized for tasks that require high throughput and can be parallelized, such as matrix calculations, image processing, and machine learning.

The primary objective of this work is to implement the CW algorithm for solving CVRP on modern GPU platforms and to analyse the efficiency of the algorithm in comparison with the CPU-based counterpart.

Even though several developments of the CW algorithm, aimed at improving its effectiveness and extensions to manage variants of the CVRP have been

proposed in the scientific literature (Borčinová, 2022; Nurcahyo, Irawan and Kristanti, 2023; Tunnisaki and Sutarman, 2023), in this work we focus on the parallel implementation of the basic CW algorithm, investigating the execution times on the GPU, and exploring the potential for application in its enhanced versions. Specifically, we analysed the steps of the CW algorithm, studying the execution times on both CPU and GPU for each step. Our analysis aims to determine which steps benefit the most from a GPU implementation and to quantify the performance improvements achieved.

In the majority of the published papers, the proposed GPU-based approaches implement existing metaheuristics into parallel approaches, to enhance execution speed, by utilizing graphics processors to handle specific segments of the code. For instance, Benaini and Berrajaa (2018) introduced an evolutionary genetic algorithm, capable of producing viable solutions for up to 3000 nodes in dynamic VRP, an extended problem variant, in which requests can occur at later times. The authors claim that their implementation is efficient and can exploit the parallelism and power of the GPU. Similarly, in Abdelatti and Sodhi (2020), the proposed methodology involves parallelizing a genetic algorithm, while incorporating local search strategies.

In Diego et al. (2012), the approach is based on Ant Colony Optimization, which determines the solution to the problem by imitating the behaviour of certain insects in nature. A different approach is developed in Luong, Melab and Talbi (2013), which is based on local search metaheuristics, i.e., on the iterative movement from one solution to another in a given neighbourhood. In Benaini, Berrajaa and Daoudi (2016), a GPU implementation of a heuristic to solve VRPs with both single and multiple depots is presented. The proposed approach computes an initial solution in parallel, which is improved iteratively. The multi-depot case is solved by decomposing the problem into several independent basic VRPs, solved in parallel. Unlike our approach, which directly optimizes the individual steps of the CW algorithm, the solution strategy presented by the authors reffered to utilizes the GPU to generate multiple initial solutions in parallel and iteratively improve them, but it relies on the standard version of CW. In Benaini and Berrajaa (2016), the authors develop an implementation of an algorithm directly on the GPU. This implementation dynamically inserts requests into various routes, and the advantage of using the GPU over the CPU counterpart implementation is also analysed.

Subsequently, in Benaini, Berrajaa and Daoudi (2017) the same authors present an extension, which is meant to handle the multi capacity VRP on the GPU.

Among more recent works, Yelmewad and Talawar (2021) use GPU to improve the performance of the local search heuristics.

To the best of our knowledge, no CUDA-based approaches aimed at imple-

menting the CW algorithm on the GPU have been proposed in the scientific literature.

The remainder of the paper is structured as follows: in Section 2, we give a detailed explanation of the approach adopted for the parallel development of the CW algorithm. Subsequently, we present the results of the comparison on benchmark instances. Finally, the paper concludes with a discussion of findings and potential future developments.

2. A GPU implementation of the CW algorithm

The proposed algorithm implementation optimizes CW heuristic by utilizing the CUDA platform. It follows steps that are similar to those of the sequential heuristic, but it executes them in parallel, harnessing the substantial computational power of the GPU.

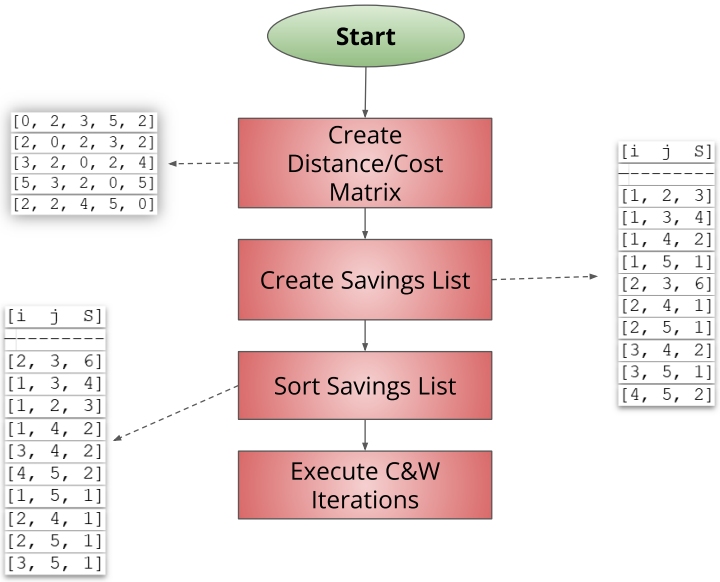

Figure 1. Principal flow

In order to explain how the CW algorithm works, let us consider a simple instance, made up of six nodes in the Euclidean space, with the following coordinates: [0, 0], [1, 1], [2, 2], [3, 3], [-1, 1], [-1 , -2]. The depot is located at the position [0,0]. The costs, i.e., the distances, are then calculated on the basis of

these coordinates. Let us assume that the corresponding demand vector is [50, 50, 50, 25, 25], and the vehicle capacity is 100.

The first operation to be carried out involves calculation of the cost matrix (see Fig. 1, box "Create Distance / Cost Matrix"). In this case, but it could easily be generalized to other metrics, the cost is given by the Euclidean distance ($d(N_i, N_j)$) between the nodes i and j, $N_i = (x_i, y_i)$, $N_j = (x_j, y_j)$:

$$C = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{1}$$

This matrix is created using a CUDA kernel, which takes care of calculating each of the elements in parallel. If the number of nodes is N, NxN threads are launched in parallel for computing this matrix. Since in this case the problem is symmetric, i.e. the cost of movement from node i to node j is equal to the cost from node j to node i, the related cost matrix will also be symmetric. In practice, in parallel, each thread created takes care of calculating a certain cell of the matrix using the formula (1).

The second step entails the creation of the Savings List (refer to Fig. 1, box "Create Savings List"), which stores the savings obtained when a given transport to node i is done by starting from another node j, rather than from the depot d of coordinates D.

In fact, if initially it is assumed that all deliveries are made from the depot (considering round trip), the cost is determined as follows:

$$C_{tot} = 2 * \sum_{i=1}^{n} d(D, N_i). \tag{2}$$

Therefore, if it is possible to make a delivery starting from a node j rather than restarting from the depot, the following savings will be obtained for the i^{th} delivery:

$$s(i, j) = d(D, N_j) + d(D, N_i) - d(N_i, N_j). \tag{3}$$

In practice, it is possible to save the cost of two trips, the one between the depot and node i, and another, between the depot and the node j, but with adding the cost of trip between node i and node j.

The savings list is computed using a CUDA kernel that consists of a large number of parallel threads. Each thread is responsible for computing the savings value for a specific pair of nodes i and j. The number of threads is equal to the

number of possible links between the nodes. Therefore, excluding the links to the depot, it is given by:

$$Link_{i,j} = N * (N-1)/2. \tag{4}$$

Each of these threads calculates one entry of the savings list, using the formula (3).

Before starting the iterations of the CW algorithm, this list is reordered according to decreasing savings values, as these nodes are processed first (Fig. 1, box "Sort Savings List"). For this standard operation of reordering the elements of a list, we used an already existing, very efficient open source library (see: https://cupy.dev/).

At this point, the main iteration of the CW algorithm is started, which processes the elements of the ordered list of savings, one after the other.

We employed a parallel variant of the CW algorithm, which simultaneously evaluates all of the routes. Alternatively, a sequential version, analysing one path at a time, is available, but it is less commonly utilized.

The steps are as follows:

1. If neither of the two nodes i,j are assigned, a new route is created.
2. Otherwise, the constraints are verified: in this work only the vehicle capacity restrictions are considered, even though it is possible to easily modify the approach to consider time windows or incompatibilities between goods constraints.
3. If the constraints are satisfied, the unassigned node is added to the route, or the two routes are merged.

In order to better clarify the behaviour of the proposed algorithm, let us consider the example introduced before. Initially, the link (2,3) associated with the largest savings, is considered. Since both nodes are unassigned, a new tour {0, 2,3, 0} is created with the load of $50 + 50 = 100$, the maximum 100 capacity of the vehicle has been reached. Links (1,3) and (1,2) are then processed, but these two links cannot be added to the current tour {0, 2,3, 0}, since one of the two nodes is already present and the vehicle is already full. Thus, the two links are rejected.

The iteration proceeds with the link (1,4), and, similarly to the first step, a new tour {0,1,4,0} with the load of 75 is created.

Then, the link (3,4) is discarded, since the two nodes are already included in the tours. The link (4,5), respecting all conditions, is added to the second tour, which therefore becomes {0,1, 4, 5,0}, with the load of 100. The other links (1,5), (2,4), (2,5) and (3,5) are all discarded, for the same reasons that have been mentioned above. The final solution is therefore characterized by the two routes: $\{0, 2, 3, 0\}, \{0, 1, 4, 5, 0\}$.

3. Computational results

The code was implemented in Python for both the CPU version and the version utilizing CUDA. The development environment used is COLAB, an on-line platform, offered by Google for the rapid development of Python-based software.

This platform allows you to use a 'Tesla T4' type GPU for free, but with a maximum RAM for the CPU of 12.7 GB and 15 GB for the GPU.

The NUMBA library was used to implement the algorithm, this library providing a set of functions, optimized for executing code, compiled directly in the CUDA platform.

Table 1. Characteristics of instances

Data Set	Min # Nodes	Max # Nodes
Axx	5	130
Mxx	101	200
Xxx	167	237
Li_xx	561	1201
Belgium	6001	16001

In order to assess the performance of the proposed approach, computational experiments have been carried out by considering the following CVRP benchmark instances: the Axx (Augerat et al., 1995), the Mxx (Christofides, 1979), the Xxx (Uchoa et al., 2017), the Li_xx and the *Belgium* sets (Arnold, Gendreau and Sörensen, 2019).

The reason behind this choice is to include easy-to-solve instances (up to 237 nodes), for assessing the overhead of the parallel algorithm, and more complicated cases (up to 16.000 nodes) to evaluate the effectiveness of the proposed approach.

Their characteristics are detailed in Table 1, which includes information on the minimum and maximum number of nodes for each dataset.

For each test problem, the execution times for each step of the algorithm (see Fig. 1) are detailed in Table 2: this table aims to isolate and show the impact of parallelization using Python for both CPU and GPU implementation.

The first column of the matrix gives the name of the tested instance, whereas in the subsequent columns, for the two approaches based on CPU and GPU, the execution times of Create Distance/Cost Matrix (i.e., Matrix), Create Savings List (i.e., List), Sorting Savings List (i.e., Ordering) and Apply CW Algorithm (i.e., ACW) steps are reported. The final column (Time) represents the total

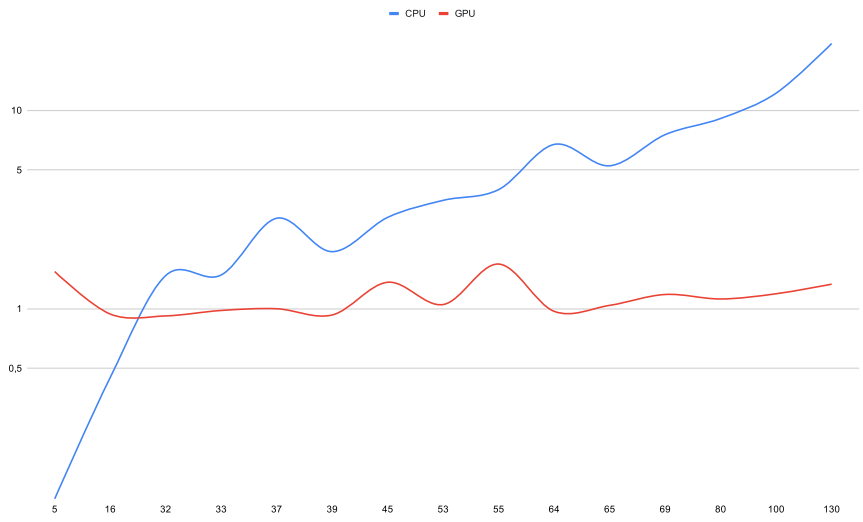

Figure 2. Computational results collected on the Axx set: CPU vs CUDA

execution time for each instance, calculated as the sum of the times for the four steps.

Due to memory constraints in the COLAB environment, it was not possible to run the Belgium instances on the CPU. Consequently, in Table 2 only the GPU execution times are provided.

The collected computational results clearly underline that the GPU implementation of the CW algorithm is significantly faster than the CPU implementation, expecially for the large-size instances. This is due to the parallel processing capabilities of GPUs, which enable efficient execution of the algorithm's computational steps.

The first sets of instances (i.e., those proposed by Augerat et al., 1995) are considerably smaller and more straightforward to be solved, when compared to the other considered sets. As expected, the speedups achieved by the GPU implementation are comparatively smaller for these instances.

However, the performance improvements of the GPU implementation compared to the CPU version become more significant for the latest set of test problems under consideration. In effect, the real potential of the GPU implementation becomes more evident in these latter instances, characterized by a high number of nodes.

Table 2. Computational results for Axx, Mxx, Xxx, Li_xx and the Belgium sets.

Instance	CPU					GPU				
	Matrix	List	Ord.	ACW	Time	Matrix	List	Ord.	ACW	Time
A-n5-k5	0.05	0.01	0.01	0.05	0.12	0.31	0.23	0.37	0.63	0.54
A-n16-k5	0.23	0.10	0.02	0.10	0.45	0.12	0.09	0.25	0.48	0.94
A-n32-k5	0.73	0.30	0.09	0.35	1.47	0.12	0.10	0.27	0.43	**0.92**
A-n33-k6	0.71	0.32	0.08	0.37	1.48	0.15	0.10	0.28	0.45	**0.98**
A-n37-k5	1.36	0.57	0.17	0.76	2.86	0.16	0.12	0.25	0.47	**1.00**
A-n39-k5	0.92	0.40	0.12	0.50	1.94	0.16	0.11	0.24	0.42	**0.93**
A-n45-k7	1.42	0.66	0.16	0.65	2.89	0.23	0.15	0.44	0.54	**1.36**
A-n53-k7	1.63	0.82	0.24	0.83	3.52	0.16	0.14	0.30	0.45	**1.05**
A-n55-k9	1.74	1.07	0.26	0.92	3.99	0.36	0.24	0.30	0.78	**1.68**
A-n64-k9	3.93	1.17	0.36	1.28	6.74	0.15	0.11	0.26	0.45	**0.97**
A-n65-k9	2.44	1.09	0.44	1.29	5.26	0.19	0.13	0.29	0.43	**1.04**
A-n69-k9	4.36	1.29	0.40	1.50	7.55	0.15	0.11	0.40	0.52	**1.18**
A-n80-k10	5.16	1.74	0.55	1.66	9.11	0.15	0.11	0.42	0.44	**1.12**
A-n100-k10	5.76	2.55	1.03	2.89	12.23	0.20	0.12	0.39	0.48	**1.19**
A-n130-k10	10.79	4.61	1.61	4.74	21.75	0.21	0.12	0.48	0.52	**1.33**
M-n121-k7	8.48	4.11	1.25	4.22	18.06	0.33	0.14	0.51	0.56	**1.54**
M-n101-k10	5.56	2.78	0.82	2.66	11.82	0.30	0.18	0.60	0.60	**1.68**
M-n151-k12	14.15	6.54	2.12	6.68	29.49	0.44	0.18	0.47	0.58	**1.67**
M-n200-k16	22.66	11.88	4.01	11.33	49.88	0.30	0.20	0.57	0.81	**1.88**
M-n200-k17	23.63	11.96	3.76	10.83	50.18	0.25	0.15	0.49	0.56	**1.45**
X-n167-k10	19.59	9.56	3.41	12.42	44.98	0.25	0.22	0.52	0.63	**1.62**
X-n172-k51	20.21	9.66	3.61	10.38	43.86	0.24	0.18	0.49	0.65	**1.56**
X-n176-k26	20.67	10.19	3.87	9.52	44.25	0.23	0.18	0.49	0.60	**1.50**
X-n181-k23	21.63	10.67	4.08	9.79	46.17	0.23	0.22	0.49	0.64	**1.58**
X-n186-k15	39.21	34.03	10.33	28.31	111.88	0.31	0.28	0.57	0.80	**1.96**
X-n195-k51	62.92	189.58	21.46	32.73	306.69	2.45	0.37	0.59	0.83	**4.24**
X-n200-k36	63.97	24.06	9.81	25.19	123.03	0.32	0.29	0.59	0.83	**2.03**
X-n204-k19	71.41	23.50	4.85	11.69	111.45	0.28	0.18	0.62	0.55	**1.63**
X-n209-k16	31.32	13.04	5.18	12.30	61.84	0.24	0.17	0.55	0.58	**1.54**
X-n214-k11	26.33	14.49	5.48	13.33	59.63	0.27	0.22	0.52	0.59	**1.60**
X-n219-k73	27.78	130.61	5.78	15.97	180.14	0.26	0.17	0.52	0.65	**1.60**
X-n223-k34	31.15	14.27	6.13	14.90	66.45	1.00	0.23	0.51	0.57	**2.31**
X-n228-k23	32.21	16.34	6.21	15.36	70.12	0.26	0.22	0.51	0.54	**1.53**
X-n233-k16	30.56	27.68	6.54	14.95	79.73	0.32	0.26	0.59	0.79	**1.96**
X-n237-k14	37.17	17.13	6.56	17.33	78.19	0.23	0.18	0.55	0.57	**1.53**
Li21	181.12	263.51	34.70	102.01	581.34	0.36	0.32	0.70	0.54	**1.92**
Li22	205.92	435.46	38.30	125.53	805.21	0.29	0.23	0.57	0.55	**1.64**
Li23	229.87	286.78	46.37	138.86	701.88	0.31	0.27	0.58	0.58	**1.74**
Li24	308.28	639.12	60.31	170.37	1178.08	0.38	0.32	0.66	0.62	**1.98**
Li25	332.93	516.36	59.70	207.97	1116.96	0.38	0.43	0.77	0.81	**2.39**
Li26	778.05	820.81	118.23	367.46	2084.55	0.60	0.49	1.05	0.74	**2.88**
Li27	838.78	847.60	119.50	422.81	2228.69	0.60	0.51	1.11	0.73	**2.95**
Li28	465.86	586.46	90.15	261.37	1403.84	0.33	0.43	1.67	0.55	**2.98**
Li29	519.94	1026.65	125.59	324.30	1996.48	0.68	0.57	1.28	0.58	**3.11**
Li30	643.86	761.85	133.52	373.53	1912.76	0.71	0.67	1.41	0.56	**3.35**
Li31	722.20	1131.88	151.75	499.56	2505.39	0.87	0.80	1.64	0.85	**4.16**
Li32	1677.61	2097.13	289.01	1121.22	5184.97	0.97	0.92	1.81	0.84	**4.54**
A1	-	-	-	-	-	17.09	18.84	36.42	37.48	109.83
A2	-	-	-	-	-	8.21	22.33	34.83	35.76	101.13
B1	-	-	-	-	-	45.57	103.33	161.93	162.91	473.74
B2	-	-	-	-	-	52.15	117.02	180.34	181.36	530.87

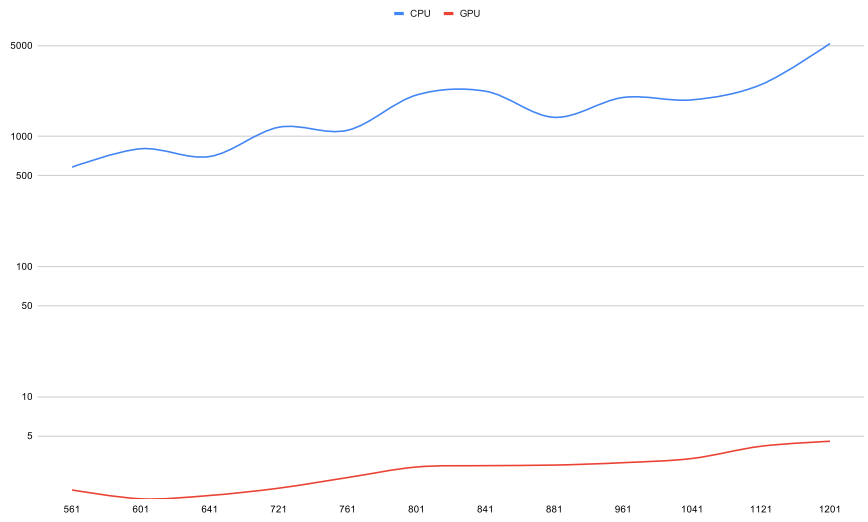

Figure 3. Computational results collected on the Li_xx set: CPU vs CUDA

Figures 2 and 3 show the speedups of the GPU version compared to the CPU one, for the simplest instances (Axx) and for the more complex ones (Li_xx), for which we also have the execution times on CPU. The horizontal axis represents the number of nodes present in the instances, while the vertical axis indicates the execution time (in milliseconds) per instance.

The logarithmic scale employed in the graph effectively highlights this exponential relationship between the size of the problem and GPU speedup. It is evident that the larger the number of nodes, the more significant the performance advantage of GPUs becomes. This observation underscores the suitability of GPUs to handle computationally intensive tasks more efficiently than CPUs for applications that demand high computational throughput, such as the CW alghoritm.

4. Conclusions

This study analyzed the execution times of various steps of the CW algorithm, which is often used as a basic heuristic for solving VRP problems. Each of these steps was implemented on the CPU and GPU, and the execution times were compared in detail. The study shows how all steps can be accelerated using the GPU and the results demonstrate that the GPU implementation of the CW

algorithm is significantly faster than the CPU version, especially for large-size instances. This is due to the parallel processing capabilities of GPUs, which enable efficient execution of the algorithm computational steps.

The study results suggest that GPU implementation can significantly improve the performance and scalability of CVRP solvers, particularly in the first three phases of the algorithm.

Therefore, this result can be exploited in enhanced versions of the CW algorithm. In fact, since the performance of the individual steps has been analyzed separately, researchers can use these findings as a reference to determine which steps are beneficial to perform on a GPU.

Moreover, by enabling a substantial acceleration in the search for a CVRP solution, albeit not necessarily optimal, this approach can be integrated into other heuristics to quickly provide an initial solution. It can also serve as the basis for an iterative method, aimed at quickly exploring the solution space.

Future research could focus on extending the applicability of GPU-accelerated CW algorithms to more complex VRP variants and exploring their potential for solving related optimization problems across various domains.

References

Abdelatti, M.F. and Sodhi, M.S. (2020) An improved gpu-accelerated heuristic technique applied to the capacitated vehicle routing problem. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO '20*. Association for Computing Machinery, New York, NY, USA, 663–671.

Accorsi, L. and Vigo, D. (2021) A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems. *Transportation Science* **55**(4), 832–856.

Arnold, F., Gendreau, M. and Sörensen, K. (2019) Efficiently solving very large-scale routing problems. *Computers & Operations Research* **107**, 32–42.

Augerat, P., Belenguer, J.M., Benavent, E., Corberan, A., Naddef, D. and Rinaldi, G. (1995) Computational results with a branch and cut code for the capacitated vehicle routing problem. Tech. rep., Research report RR949-M. ARTEMIS-IMAG, France.

Benaini, A. and Berrajaa, A. (2016) Solving the dynamic vehicle routing problem on gpu. In: *2016 3rd International Conference on Logistics Operations Management (GOL)*, IEEE, 1–6.

Benaini, A. and Berrajaa, A. (2018) Genetic algorithm for large dynamic vehicle routing problem on gpu. In: *2018 4th International Conference on Logistics Operations Management (GOL)*, IEEE, 1–9.

Benaini, A., Berrajaa, A. and Daoudi, E.M. (2016) Solving the vehicle routing problem on gpu. In: A. El Oualkadi, F. Choubani, A. El Moussati, eds., *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*. Springer International Publishing, Cham, 239–248.

Benaini, A., Berrajaa, A. and Daoudi, E.M. (2017) Parallel implementation of the multi capacity vrp on gpu. In: Á. Rocha, M. Serrhini, C. Felgueiras, eds., *Europe and MENA Cooperation Advances in Information and Communication Technologies*. Springer International Publishing, Cham, 353–364.

Borčinová, Z. (2022) Kernel search for the capacitated vehicle routing problem. *Applied Sciences* **12**(22).

Christofides, N. (1979) *Combinatorial Optimization*. A Wiley-Interscience publication. Wiley

Clarke, G. and Wright, J.W. (1964) Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research* **12**(4), 568–581.

Diego, F.J., Gómez, E.M., Ortega-Mier, M. and García-Sánchez, Á. (2012) Parallel cuda architecture for solving de VRP with ACO. In: S.P. Sethi, M. Bogataj, L. Ros-McDonnell, eds., *Industrial Engineering: Innovative Networks*. Springer London, London, 385–393.

Hijma, p., Heldens, S., Sciocco, A., van Werkhoven, B. and Bal, H.E. (2023) Optimization Techniques for GPU Programming. *ACM Comput. Surv.* **55**(11), 1–81.

Laporte, G. (1992) The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* **59**(3), 345–358.

Liu, F., Lu, C., Gui, L., Zhang, Q., Tong, X. and Yuan, M. (2023) Heuristics for vehicle routing problem: A survey and recent advances. arXiv:https://arxiv.org/abs/2303.04147

Luong, T.V., Melab, N. and Talbi, E.G. (2013) Gpu computing for parallel local search metaheuristic algorithms. *IEEE Transactions on Computers* **62**(1), 173–185.

Nurcahyo, R., Irawan, D.A. and Kristanti, F. (2023) The effectiveness of the Clarke & Wright savings algorithm in determining logistics distribution routes (case study pt.xyz). *E3S Web of Conferences* **426**. EDP Sciences.

Tunnisaki, F. and Sutarman, F. (2023) Clarke and Wright savings algorithm as solutions vehicle routing problem with simultaneous pickup delivery (vrpspd). *Journal of Physics: Conference Series* **2421**(1), 012045.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T. and Subramanian, A. (2017) New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research* **257**(3), 845–858.

Yelmewad, P. and Talawar, B. (2021) Parallel version of local search heuristic algorithm to solve capacitated vehicle routing problem. *Cluster Computing* **24**(4), 3671–3692.