

A HYBRID GENETIC ALGORITHM FOR THE QUADRATIC ASSIGNMENT PROBLEM ON GRAPHICS PROCESSING UNITS

Erdener ÖZÇETİN^{1,*}, Gürkan ÖZTÜRK¹

¹Endüstri Mühendisliği Bölümü, Mühendislik Fakültesi, Anadolu Üniversitesi 26555 Tepebaşı, Eskişehir

ABSTRACT

In this paper, a hybrid genetic algorithm is proposed for the quadratic assignment problem. The most time-consuming parts of the proposed algorithm are the calculation of objective function values and the local search operator. Therefore, the parallelization and implementation on graphics processing units of these parts was addressed. This parallel algorithm and its sequential version have been tested and compared for 49 instances in the literature. The best-known solutions were obtained for 34 of these instances. Computational experiments show that the proposed algorithm is capable of providing good quality solutions in a short time. Indeed, it can be observed that the parallel algorithm works up to 51 times faster --17 times faster on average-- than the sequential algorithm.

Keywords: Quadratic assignment problem (QAP), parallel programming, graphics processing units (GPU), CUDA

KARESEL ATAMA PROBLEMİ İÇİN GRAFİK İŞLEM BİRİMLERİ ÜZERİNDE TASARLANMIŞ BİR MELEZ GENETİK ALGORİTMA

ÖZET

Bu çalışmada karesel atama probleminin çözümü için melez bir genetik algoritma önerilmiştir. Önerilen algoritmanın en zaman alıcı bölümleri amaç fonksiyonunun hesaplanması ve yerel arama operatörüdür. Bu nedenle algoritmanın söz konusu bölümlerinin paralelleştirilmesi ve grafik işlem birimleri üzerinde uygulanması üzerinde durulmuştur. Algoritmanın seri ve paralel versiyonu 49 adet literatür problemi üzerinde test edilmiş ve karşılaştırmalar yapılmıştır. Test edilen literatür problemlerinden 34'ü için bilinen en iyi sonuçlara ulaşılmıştır. Deneysel çalışmalar önerilen algoritmanın kısa sürede etkin sonuçlar verebildiğini ortaya koymuştur. Önerilen paralel algoritmanın ortalama 17 kat olmak üzere 51 kata kadar seri algoritmaya göre hızlı çalıştığı raporlanmıştır.

Keywords: Karesel atama problemi, paralel programlama, grafik işlem birimleri, CUDA

1. INTRODUCTION

The quadratic assignment problem (QAP) is a well-known combinatorial optimization problem introduced by Koopmans and Beckmann [1]. The problem has a wide range of applications, including facility planning [2], backboard wiring [3] and aircraft gate assignment [4].

It has been demonstrated that the QAP is *NP-Hard* [5]. As the QAP is a facility planning problem, n facilities have to be assigned to n candidate locations by minimizing the cost function. Let $D_{n \times n}$ be the distance matrix such that d_{ij} denotes the distance between i^{th} and j^{th} locations, and let $F_{n \times n}$ be a flow matrix such that f_{kl} is the flow between facilities k and l . For permutation-based representation of solution, any π interprets the assignments. Let, π_1 be $\{4 1 3 2\}$. For π_1 , facility 4 is assigned to location 1, facility 1 is assigned to location 2, facility 3 is assigned to location 3 and facility 2 is assigned to location 4. The objective is to find the best permutation-based solution (π^*) that minimizes the cost function given by

$$C(\pi) = \sum_{i=1}^n \sum_{j=1}^n d_{ij} f_{\pi(i)\pi(j)}. \quad (1)$$

* Corresponding Author: eozcetin@anadolu.edu.tr

Since the QAP is *NP-Hard*, obtaining optimal solutions using exact methods may not even be doable (for detailed information see [6]). For this reason, researchers have proposed meta-heuristics. To date, genetic algorithms (GA) (e.g. [7, 8]), Tabu search (TS) (e.g. [9, 10, 11, 12]), simulated annealing (SA) (e.g. [13, 14, 15]) and ant colony optimization (ACO) (e.g. [16, 17]) have been proposed. These research shows that even obtaining good quality solutions may very well be time-consuming when large-scale instances of the problem are considered.

Graphics processing units (GPUs) are processors on the chipset of video cards that were originally designed for data visualization. Due to user-friendly software, GPU computational capabilities became available for researchers. The introduction of compute unified device architecture (CUDA) by NVIDIA in 2006 can be considered a turning point in the usage of GPUs in the applied sciences. Since then, thousands of publications have been added to the literature from a variety of disciplines.

Programming on GPUs also drew the attention of researchers working on combinatorial optimization. It has become a widespread practice to develop meta-heuristics on GPUs to reduce computation times for combinatorial optimization problems. A parallel GA was proposed on GPUs for the QAP; the observed computation times on GPUs were 3.12 times faster than the central processing units (CPUs) [18]. In another study, an ACO algorithm is applied for the QAP with a local search procedure [19]. Since local search is the most time-consuming part of the algorithm, parallelization of this procedure on GPUs is focused. Speedups of up to 24.6 times were attained in this study. Luong et al. also considered the QAP, presenting a new methodology to design and implement hybrid evolutionary algorithms on GPUs [20]. They attained up to 14.1 times speedups. Zhu et al. [21] proposed a parallel TS algorithm for QAP on GPUs. Recently, a multi-start TS algorithm was proposed on GPUs to solve the QAP [22]. Speedups of up to 50 times were generated based on six-core CPUs for symmetric instances.

There are also several studies in the literature related to other combinatorial optimization problems whose solution involved meta-heuristics on GPUs. Cecilia et al. [23] developed an ACO algorithm for the traveling salesman problem (TSP) and reported speedups of up to 20 times. Delevacq et al. [24] also studied the TSP and were able to reach speedups of 23 times compared to the sequential implementation of their ACO algorithm. Shulz [25] presented a study for solving the vehicle routing problem (VRP) based on local search algorithms. In the study, memory bottlenecks and their impact on performance are discussed, but speedup results are not mentioned. Groer et al. [26] also considered the VRP and employed local search methods. Huang et al. [27] studied the scheduling problem using a random key GA. They emphasized that chromosomes constructed via random key can be effectively implemented on GPUs. Czapinski, Barnes [28] also took up the scheduling problem and developed a TS algorithm to solve it. In the study they reported speedups of up to 89 times.

In addition to these studies, Bozejko [29] proposed a new parallel objective function determination method for the flexible job shop scheduling problem. They obtained an average 45 times speedup by using the shared memory-based version of the parallel algorithm and a 27 times speedup when using the global memory-based version. Furthermore, Kneusel [30] studied curve fitting with particle swarm optimization on GPUs.

In this paper, a hybrid GA is proposed to solve the QAP. This algorithm is implemented in a parallel manner on GPUs and sequentially on CPUs. In the parallel implementation, the algorithm's objective function evaluation and local search procedure are parallelized on GPUs using the CUDA development environment. Our goal is to develop a robust and effective algorithm leading to good feasible solutions for instances of the QAP and to obtain speedups on computation times with the parallel implementation on GPUs.

The rest of this paper is structured as follows: In Section 2, we give information about GPUs, CUDA and the use of GPUs for combinatorial optimization problems. Then, we explain the new algorithm and

its implementation details in Section 3. The computational results and some comparisons are presented and discussed in Section 4 with concluding remarks in Section 5.

2. HIGH-PERFORMANCE COMPUTING ON GPUS FOR COMBINATORIAL OPTIMIZATION PROBLEMS

The demand of computer game players and producers has led to the development of today’s graphics card technology, resulting in cards with hundreds of cores. Massive data parallelization is used to cope with the challenges encountered in the computation of 3D visualization for computer games, CAD programs, movies, and other applications. This computing power can also provide opportunities for the applied sciences, and well-organized parallel algorithms can offer tremendous speedups with GPUs.

Matrix multiplication is one of the most oft-mentioned computational cases for parallel implementation on GPUs [31]. We also worked on the same case, to demonstrate and emphasize GPUs computational power by means of speedups. In the first step of our implementation, two matrices were randomly generated using decimal numbers. After implementations on both CPUs and GPUs, comparisons were made by means of speedups. For the multiplication of matrices $A_{4096 \times 4096}$ and $B_{4096 \times 4096}$, the computation time was almost 1440 seconds with CPUs, but only 5.1 seconds with GPUs. The speedups can be calculated as a ratio of CPU time to GPU time. All the speedups for different matrix sizes are shown in Table 1. According to the experimental results, speedups increase dramatically as matrix size grows.

Table 1. Results of matrix multiplication

Matrix Size	CPU time (s)	GPU time (s)	Speed Up
256	0.066	0.297	-
320	0.124	0.356	-
512	0.704	0.327	2.15
640	1.400	0.319	4.39
1024	6.900	0.305	22.62
1280	13.930	0.466	29.89
2048	147.750	0.906	163.08
4096	1439.330	5.100	282.22

2.1. General GPU Structure

After the first CUDA-enabled graphics card was produced, NVIDIA released five architectures for GPUs. Although the power of a GPU does not equal that of a CPU, a number of GPUs are much more powerful than CPUs, and in recent years GPU performance has increased more rapidly than that of CPUs.

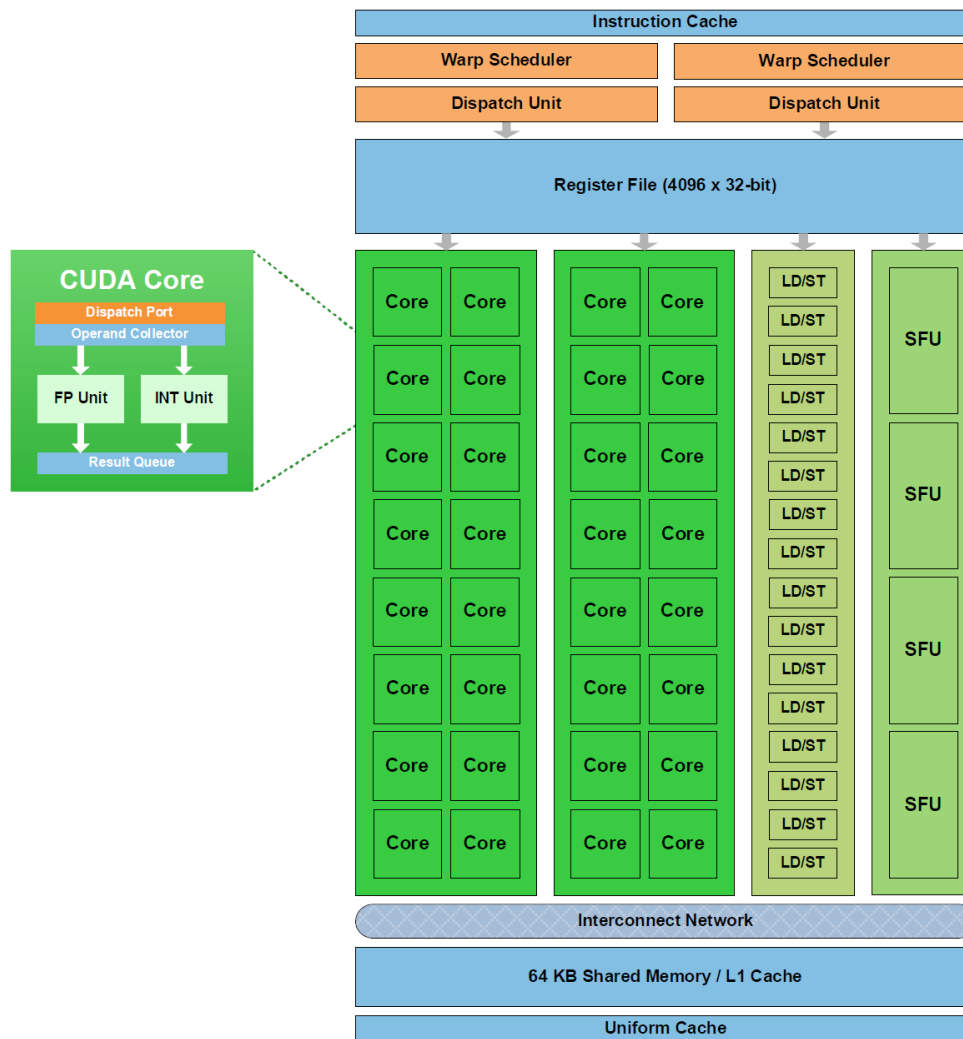


Figure 1. Fermi architecture [32]

A CUDA core is the processor on a chipset that is also called a GPU. The GTX580 graphics card designed with Fermi architecture (Figure 1) is used in this study. In this architecture, CUDA cores are assembled in a group known as a streaming multiprocessor (SM). Each SM has 32 CUDA cores and there are a total 16 SMs on the chipset. Each SM has 64 KB of on-chip memory that can be configured to have up to 48 KB of shared memory and 16 KB as L1 cache, or 16 KB as shared memory and 48 KB as L1 cache. Fermi includes a L2 cache of 768 KB memory that is shared with all SMs. Furthermore, there are four special function units (SFU) on each SM to conduct special mathematical computations and two warp schedulers (WS) to organize the threads. Each CUDA core has its own integer unit (INT) and floating point unit (FP) for integer and floating point computations.

2.2 GPU Programming Strategies

It is possible to develop programs on GPUs using various programming languages, like C, C++ and Fortran on Microsoft Visual Studio or different compilers. Some parts of these programs continue to run on CPUs, with the rest of the programs executed on GPUs using kernels. A kernel is a code block that triggers the GPUs to work. Each kernel in a program needs grid parameters for executing code on GPUs. A thread is the smallest work piece, and the code is executed in groups of 32 threads called warps. A grid contains blocks, and a block contains a set of threads. It is bound with 1024 threads per block in

Fermi. Using block or thread indices and dimensions when implementing algorithms can offer many advantages. For example, the indices can be used to compute array subscripts. Additionally, each SM can work with up to eight blocks and 1536 threads. However, there are limitations, such as restrictions using load and store units. Each SM has 16 load and store units that organize the calculation for 16 threads per clock cycle. Consequently, performance tuning on GPUs requires planning enough parallelism to occupy all the multiprocessors.

Beyond raw physical parameters such as capacity and bandwidth, memory optimization also plays a key role in higher performance. The main strategy for this issue is avoiding redundant data transfer and keeping data in the fastest available memory. There are several memory types for GPUs, and their comparisons are mainly based on speed. Registers, the private memory for each thread, are the fastest kind of memory on GPUs. Each SM on the board has its own register memory that is divided between the threads assigned to that SM. Shared memory is the second fastest kind of memory. This memory is accessible to threads within a block. Shared memory can be inefficient if not accessed properly. The global memory residing in the devices DRAM is the slowest memory type due to high latency. Efficient access to global memory is related to grid organization. Threads properly grouped into warps can tolerate long latency operations. Besides the L1 and L2 caches, Fermi also includes dedicated texture and constant memory caches to accelerate reading global memory and to send kernel argument interaction. In addition to these memories, page-locked memory guarantees the specified memory in RAM to transfer data between CPUs and GPUs (You can find detailed information about CUDA in [31]).

3. THE PARALLEL HYBRID GENETIC ALGORITHM FOR THE QAP

In this section, our hybrid GA approach is presented with all operators. Then, the parallelized operators for GPU implementation are described.

3.1. GA for QAP

Here, the GA first proposed by Holland [33] with various modifications are presented. The GA is an evolutionary algorithm and this type is generally good at diversification. The main reason for hybridizing the GA with local search algorithms is to enhance it by means of intensification. In our approach, we used genetic operators with some modifications to avoid a premature convergence problem. Moreover, we hybridized our GA with a local search operator to optimize our solutions.

3.1.1 Encoding and initial population generation

The first step in the GA approach is to encode the solutions and generate them. In our approach, we used permutation-based solutions and generated them randomly. In a permutation, each position means the assignment of facilities to candidate locations. For instance, if there is a 4 in the first position of the π , this means the fourth facility is assigned to the first location. In our implementation, each π represents an individual for the GA, and we worked with a population of 1000 individuals according to our preliminary computational tests. After generating all these solutions randomly, each individual's objective function value was calculated according to Equation 1.

3.1.2. Selection

The selection operator is designed to select the parents to apply the crossover operator. This operator is the first operator for the loop of each iteration. Basically, a modified tournament selection procedure is used for this operator. In the literature, a typical tournament selection criterion is to always choose the fittest individual among the candidate individuals; however, this leads to a premature convergence problem in the GA.

In our modified procedure, two individuals were chosen from the population randomly and then compared based on their objective function values. The fitter individual wins the tournament at the probability of 0.85. Otherwise the other individual wins. Our preliminary computational tests suggested that this probability was used to increase diversity in the next generation.

3.1.3. Crossover

The crossover scheme is widely reported as critical to GA performance. For the proposed algorithm, this operator works with the 80% of selected individuals and produces the next generation by combining the characteristics of both parents. The remaining individuals are directly transferred to the next generation. We considered one-point, two-point and position-based crossover techniques for our GA. Experiments indicate that position-based crossover works better than the others. Therefore this technique is used for the crossover operator.

In our implementation, consecutive individuals (parents) were included in the crossover procedure. A k parameter was selected randomly, which indicated the number of transferring genes from consecutive parents. When the first new individual (child) was constituted, the locations of the k genes were selected and taken from the first parent at random. The unassigned positions of the first child were scanned from the left to right of the second parent. The next child was constituted in a manner similar to the first one (see also Figure 2).

$k=7$	↓	↓		↓	↓	↓	↓	↓		↓		
Parent1	2	8	12	1	3	5	6	11	9	4	7	10
Parent2	4	9	5	7	10	1	3	2	6	8	11	12
Child1	2	8	4	5	3	10	6	11	9	1	7	12
Child2	4	9	8	12	10	1	3	2	6	5	11	7

Figure 2. Position based crossover

3.1.4 Mutation

After crossover, a mutation procedure was applied to all individuals. For each individual, two genes were randomly selected and the difference arising from this change was calculated. If this change improved the solution, it was always accepted. On the other hand, if the change was not valuable, it was accepted with a probability of 0.1, otherwise it was rejected. This modification serves to maintain population diversity.

3.1.5 Local search

A basic ingredient of hybridized meta-heuristics is local search, which brings significant intensification to the algorithm. Basically, this swap based operator searches for changes to gene pairs for all individuals to improve solutions. The idea behind local search is very simple. Each iteration starts with an initial solution π , then a number of neighboring solutions are obtained with the swap operator to π . It must be noted that there may be many improving solutions in the neighborhood of the current solution. Thus, we need a strategy for the acceptance of improving neighbors. The most common acceptance strategies are accepting the best improving neighbor and accepting the first improving neighbor. We utilized the accepting the first improving neighbor strategy. Therefore, if a neighbor solution π' is better than π , the process continues with π replacing π' .

If we think about an n dimensional symmetric QAP instance, there are $\frac{n(n-1)}{2}$ candidate changes for each solution. An iteration of this operator consists of evaluating the neighbors with respect to the difference between the current and candidate solutions (see Equation 2). If this difference is less than zero, then the change is accepted. Otherwise, it is rejected, and the next candidate is considered.

$$\delta = \sum_{t=1}^n \sum_{t \neq a, t \neq b} (d_{at} - d_{bt})(f_{\pi(b)\pi(t)} - f_{\pi(a)\pi(t)}) \quad (2)$$

When this operator ends its search, the elitism operator keeps the best solution in the population, and the iteration finishes. The pseudo-code of our hybridized GA can be seen in Figure 3.

```

initialize the population ;
while termination condition is not satisfied do
    Selection      modified tournament selection
    Crossover      position based crossover
    Mutation       modified swap mutation
    Local Search   swap based method
end
    
```

Figure 3. Pseudo-code for the proposed algorithm

3.2. Parallel Implementation of the Algorithm on GPUs

As mentioned in Section 1, the main focus of this study was to propose a fast and robust algorithm that provides efficient solutions for the QAP.

Many algorithms are well-designed with the powerful combination of GPUs and CPUs. The most time-consuming parts of our hybridized GA were the calculation of objective function values and the local search operator. Therefore, the parallelization and implementation on GPUs of these parts was addressed. The remaining parts of the algorithm concentrate on the CPUs.

To begin with, input data $D_{n \times n}$ and $F_{n \times n}$ was read and copied from host to global memory. These two matrices were available for all calculations on GPUs, until the termination criteria were reached. Using constant memory can be thought of as more efficient than global memory for storing these data, but this is not possible for large-scale instances due to the limitations of constant memory.

3.2.1. Parallel objective function evaluation

For each iteration of the algorithm, all individuals were transferred to the global memory. Then, each individual's objective function value was evaluated simultaneously. In addition, multiplication operations were handled in a parallel manner for any individual's objective function evaluation.

Specifically, we examined 3 individuals in the population. Let, π_1, π_2 and π_3 be $\{1\ 2\ 3\ 4\}$, $\{4\ 1\ 3\ 2\}$ and $\{1\ 3\ 4\ 2\}$ respectively. Any π interpreted the assignments. For π_3 , facility 1 was assigned to location 1, facility 3 was assigned to location 2, facility 4 was assigned to location 3 and facility 2 was assigned to location 4. For any individual's evaluation as a symmetric QAP instance, there existed $\frac{n(n-1)}{2}$ multiplication operations. All these multiplications were handled simultaneously for all individuals and put in the vector A using thread and block indices (see Figure 4). Then, A was summed up, and all individuals' objective function values were calculated. After all corresponding fitness values were calculated, all individuals were kept in the global memory for parallel local search. This means that there was no memory operation at the end of this evaluation or at the beginning of the parallel local search.

As previously mentioned, there were 1000 individuals in the population. Initial experiments showed that the best choice for CUDA launch parameters was to group threads into 8 blocks of 125 threads.


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}
	f_{12}	f_{13}	f_{14}	f_{23}	f_{24}	f_{34}	f_{41}	f_{43}	f_{42}	f_{13}	f_{12}	f_{32}	f_{13}	f_{14}	f_{12}	f_{34}	f_{32}	f_{42}
A	$d_{12}f_{12}$	$d_{13}f_{13}$	$d_{14}f_{14}$	$d_{23}f_{23}$	$d_{24}f_{24}$	$d_{34}f_{34}$	$d_{12}f_{41}$	$d_{13}f_{43}$	$d_{14}f_{42}$	$d_{23}f_{13}$	$d_{24}f_{12}$	$d_{34}f_{32}$	$d_{12}f_{13}$	$d_{13}f_{14}$	$d_{14}f_{12}$	$d_{23}f_{34}$	$d_{24}f_{32}$	$d_{34}f_{42}$
Obj.	$2(A_1 + A_2 + A_3 + A_4 + A_5 + A_6)$						$2(A_7 + A_8 + A_9 + A_{10} + A_{11} + A_{12})$						$2(A_{13} + A_{14} + A_{15} + A_{16} + A_{17} + A_{18})$					

Figure 4. Parallel objective function evaluation

3.2.2. Parallel local search

The local search operator is carried out on GPUs in a similar fashion to the objective function evaluation. There exists $\frac{n(n-1)}{2}$ candidate swaps forüa any individual in the population.

Instead of evaluation in a sequential manner, the swap of the genes at the same position for all individuals is checked simultaneously. Again, we examined 3 individuals in the population. In Figure 5, we see an example of swap control between the first and third genes for all individuals. In addition, as mentioned above, the difference in the objective function values of neighbors is calculated using Equation 2. We implemented this function only on those GPUs that can be called from kernels. Once the entire neighborhood had been performed in parallel on GPUs, all individuals were copied back to the host. After that, a loop of our algorithm ends with the elitism operator on the CPU side.



π_1	1	2	3	4
π_2	4	1	3	2
π_3	1	3	4	2

Figure 5. Parallel local search

4. COMPUTATIONAL RESULTS

As mentioned in the introduction, there are a wide variety of GPU implementation papers in different areas of scientific computing, including a few in combinatorial optimization with meta-heuristics. In this section, we demonstrate how we gained speedups with our fair CPU and GPU implementations.

4.1. Test Environment and Instances

All experiments were performed on an NVIDIA GeForce GTX 580, which is a Fermi architecture GPU. The GPU code was compiled using CUDA 4.2. Moreover, computational experiments were conducted on a six-core Intel Core i7 3.20 GHz CPU with 32 GB of memory, running under Windows 7 (64-bit). All algorithms were implemented with C++ on Microsoft Visual Studio 2010. For the comparisons, we used some symmetric QAP instances with different sizes from QAPLIB [34]. These test instances have different origins. Therefore, the efficiency of the algorithm was demonstrated with different types of instances. TaiXXa problems are randomly generated according to a uniform distribution. In the class of NugXX and SkoXX instances, flows were randomly generated, and the distances defined as the "Manhattan" distance between grid points. EscXX instances are real-life, structured instances including problems from practical applications. These test instances and their sizes are given in Table 2.

Table 2. Test instances and their sizes

Test Instance	Size
Escherman (EscXX)	32a, 32b, 32c, 32d, 32e, 32f, 32g, 32h, 64, 128
Nugent (NugXX)	12, 14, 15, 16a, 17, 18, 20, 21, 22, 24, 25, 27, 28, 30
Taillard (TaiXX)	12a, 15a, 17a, 20a, 25a, 30a, 35a, 40a, 50a, 60a, 80a, 100a
Skorin (SkoXX)	42, 49, 56, 64, 72, 81, 90, 100a, 100b, 100c, 100d, 100e, 100f

4.2. Test Results

The results of comparisons are shown in the Tables (3, 4, 5, 6). In these tables, BK is the best known or optimal solution of the test instance. The CPU and GPU columns show the average runtime of the implementations in seconds. In order to show the performance of the proposed algorithm, HIT counts were used, which indicate how many times the best known values were reached in 20 runs. These are shown in the HIT column. Note that each speedup was calculated as a ratio of average CPU time to average GPU time. In addition, the GAP CPU and GAP GPU columns indicate the average gaps of CPU and GPU implementation, and they were obtained by using Equation 3.

$$\frac{C(\pi) - C(\pi^*)}{C(\pi^*)} \tag{3}$$

Table 3. Results of Escherman problems

Problem	BK	CPU (s)	GPU (s)	GAP CPU	GAP GPU	Hit	Speed up
Esc32a	130	176.651	6.195	0.0006	0.0007	19/20	28.52
Esc32b	168	5.493	0.229	0	0	20/20	23.99
Esc32c	642	1.553	0.154	0	0	20/20	10.08
Esc32d	200	2.360	0.155	0	0	20/20	15.23
Esc32e	2	1.558	0.153	0	0	20/20	10.18
Esc32f	2	1.554	0.150	0	0	20/20	10.36
Esc32g	6	1.553	0.152	0	0	20/20	10.22
Esc32h	438	3.216	0.158	0	0	20/20	20.35
Esc64	116	3.837	0.872	0	0	20/20	4.40
Esc128	64	14.44	6.322	0	0	20/20	2.28
Average		21.222	1.454	0.00006	0.00007		13.56

The results for Escherman's problems are given in Table 3. The number of times the best known solution was found is 20/20 for all instances except Esc32a, which was 19/20. If we focus on Esc32a, which is known as the hardest problem in this set, it took an average 176.651 seconds to reach the average gap 0.0006 (0.06%) with the CPU implementation, but only 6.195 seconds with the GPU implementation. The experimental results show an impressive reduction in execution time. The average speedup was 28.52 times for Esc32a and 13.56 times for all other problems in this set. Figure 6 shows the average solution times for both CPU and GPU runs.

Table 4. Results of Nugent problems

Problem	BK	CPU (s)	GPU (s)	GAP CPU	GAP GPU	Hit	Speed up
nug12	578	0.585	0.035	0	0	20/20	16.71
nug14	1014	1.747	0.064	0	0	20/20	27.30
nug15	1150	1.068	0.046	0	0	20/20	23.22
nug16a	1610	3.917	0.083	0	0	20/20	47.46
nug17	1732	5.522	0.154	0	0	20/20	35.79
nug18	1930	6.756	0.174	0	0	20/20	38.74
nug20	2570	5.637	0.154	0	0	20/20	36.58
nug21	2438	8.904	0.309	0	0	20/20	28.83
nug22	3596	5.247	0.189	0	0	20/20	27.80
nug24	3488	10.132	0.446	0	0	20/20	22.70
nug25	3744	16.086	0.509	0	0	20/20	31.58
nug27	5234	18.476	0.789	0	0	20/20	23.42
nug28	5166	47.685	2.734	0	0	20/20	17.44
nug30	6124	69.989	8.549	0.0005	0.0004	15/20	8.19
Average		14.411	1.017	0.00004	0.00003		27.55

Table 5. Results of Skorin problems

Problem	BK	CPU (s)	GPU (s)	GAP CPU	GAP GPU	Hit	Speed up
Sko42	15812	139.040	12.090	0.0001	0.0001	19/20	11.50
Sko49	23386	630.635	57.630	0.0012	0.0012	1/20	10.94
Sko56	34458	354.380	29.970	0.0011	0.0013	0/20	11.83
Sko64	48498	473.065	73.840	0.0006	0.0003	7/20	6.41
Sko72	66256	416.802	60.860	0.0039	0.0043	0/20	6.85
Sko81	90998	434.909	67.810	0.0017	0.0018	0/20	6.41
Sko90	115534	433.990	72.160	0.0038	0.0037	0/20	6.01
Sko100a	152002	950.996	94.940	0.0029	0.0032	0/20	10.02
Sko100b	153890	763.755	98.130	0.0019	0.0027	0/20	7.78
Sko100c	147862	879.414	93.310	0.0030	0.0031	0/20	9.42
Sko100d	149576	1177.933	129.300	0.0019	0.0022	0/20	9.11
Sko100e	149150	773.763	92.230	0.0017	0.0019	0/20	8.39
Sko100f	149036	1301.036	111.870	0.0028	0.0031	0/20	11.63
Average		671.517	76.470	0.0020	0.0022		8.95

In Table 4 and Figure 7, comparisons for Nugent problems are presented. For this set, speedups vary between 8.19 times and 47.46 times. While the average execution time for Nug28 on CPU was 47.7 seconds, it was only 2.7 seconds on GPUs. Moreover, HIT counts were 20/20 for 13 of these instances.

For Skorin instances, the problem sizes varied from 42 to 100. A stopping criterion was defined for this set to make the comparison fair. According to this criterion, the execution ended when the gap was less than or equal to 0.005 (0.5%). An 8.95 times average speedup was attained for this set. Table 5 and Figure 8 indicate the average solution times for the implementations.

It is a fact that most of the Taillard instances are still in demand for improving best known solutions. In 2012, Misevicius reported the new best known values of the objective function for instances tai50a, tai80a, and tai100a [12]. For this set, the stopping criterion was 0.02 (2%). The experimental results show that our algorithm ran on average 15.68 times faster on GPUs than the CPU version for this set (see Table 6). Figure 9 illustrates the average execution times of the two versions.

Table 6. Results of Taillard problems

Problem	BK	CPU (s)	GPU (s)	GAP CPU	GAP GPU	Hit	Speed up
Tai12a	224416	0.467	0.028	0	0	20/20	16.83
Tai15a	388214	2.967	0.164	0	0	20/20	18.15
Tai17a	491812	6.406	0.125	0	0	20/20	51.27
Tai20a	703482	126.707	5.637	0	0	20/20	22.48
Tai25a	1167256	149.951	13.999	0.0036	0.0034	5/20	10.71
Tai30a	1818146	322.371	38.664	0.0031	0.0032	5/20	8.34
Tai35a	2422002	400.977	89.797	0.0100	0.0091	1/20	4.47
Tai40a	3139370	449.663	91.118	0.0130	0.0130	0/20	4.93
Tai50a	4938796	487.824	119.537	0.0199	0.0197	0/20	4.08
Tai60a	7208572	791.636	61.517	0.0199	0.0199	0/20	12.87
Tai80a	13515450	853.236	78.646	0.0199	0.0199	0/20	10.85
Tai100a	21054656	1217.771	52.467	0.0199	0.0199	0/20	23.21
Average		400.831	45.975	0.0091	0.0090		15.68

5. CONCLUSION

In this study, we introduced a hybrid GA to solve the QAP on GPUs. Modified operators and local search were combined to diversify and intensify the search space. This algorithm is traditionally implemented on CPUs. The most time-consuming parts of the algorithm were the objective function evaluation and the local search procedure. Therefore, the parallelization of these two main parts of the algorithm was addressed. Efficient parallelization and implementation techniques were suggested for these operations on GPUs by using CUDA.

The computational results and comparisons demonstrated that the proposed method yields robust solutions for the QAP instances. Our hybridized GA found the best known solutions for 34 out of 49 benchmark problems. On average, the results were only 0.003 (0.3%) worse than the best known solutions. Furthermore, parallel implementation on GPUs works on average 17 times and up to 51 times faster than the CPU version.

Further research should focus on how other time-consuming operators, like the crossover operator, can be implemented on GPUs. In addition, research should focus on tuning up the proposed method to attain even better solution quality. The proposed algorithm can also be applied to other discrete optimization problems. The performance of the proposed algorithm with other problems will be the subject of future research.

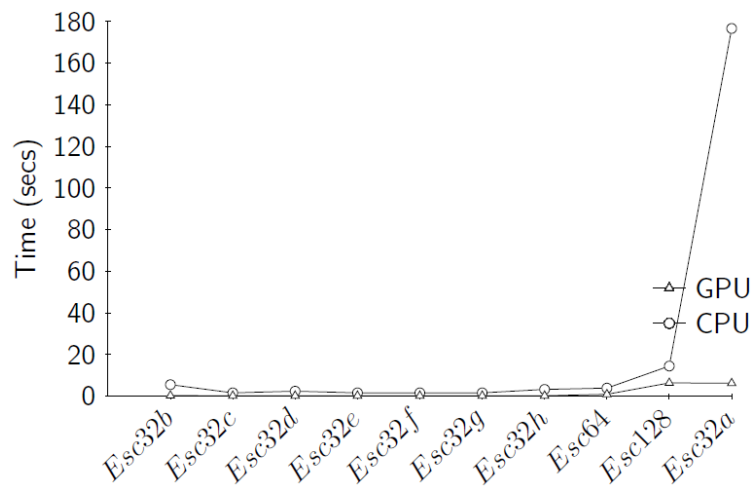


Figure 6. Escherman instances

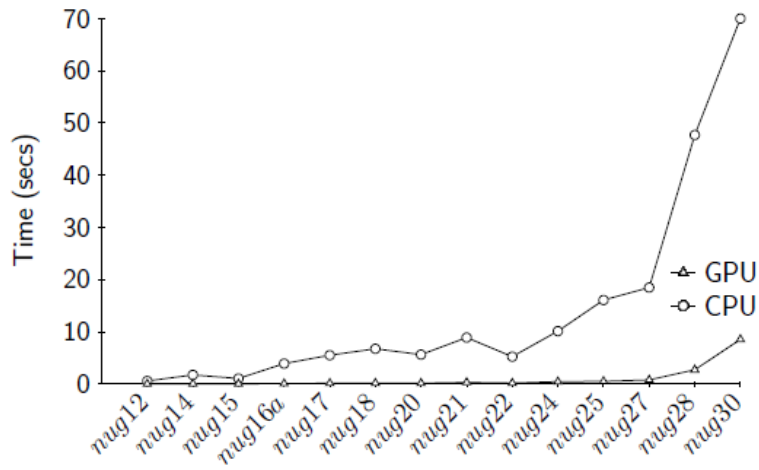


Figure 7. Nugent instances

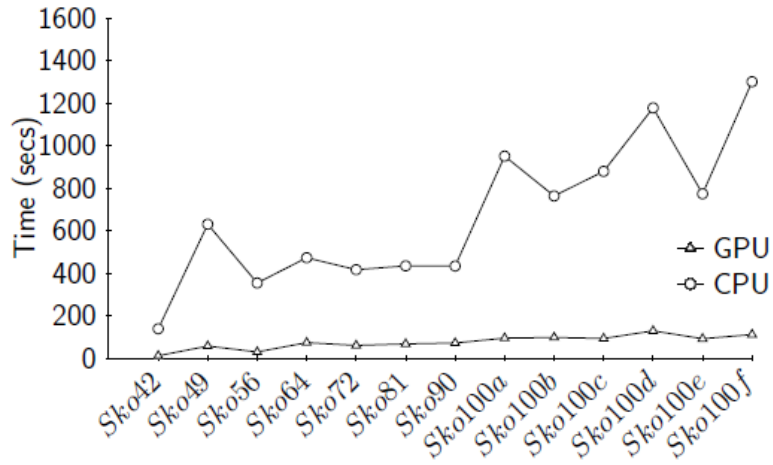


Figure 8. Skorin instances

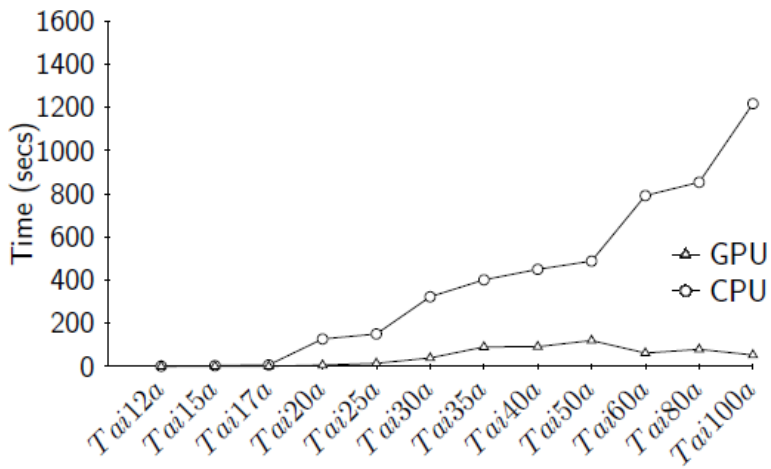


Figure 9. Taillard instances

REFERENCES

- [1] T C Koopmans and M Beckmann. Assignment problems and the location of economic activities. *Econometrica: Journal of the Econometric* 1957; 25 (1): 53–76.
- [2] A N Elshafei. Hospital layout as a Quadratic Assignment Problem. *Operations Research Quarterly* 1977; 28: 167–179.
- [3] L Steinberg. The backboard wiring problem: a placement algorithm. *SIAM Review* 1961; 3: 37–50.
- [4] A Haghani and M -C. Chen, Optimizing gate assignments at airport terminals, *Transportation Research Part A: Policy and Practice* 32 (1998) 437–454.
- [5] S Sahni and T Gonzalez. OP-Complete approximation problems. *Journal of the ACM* 1976; 23: 555-565.
- [6] Z Drezner, P M Hahn, E D Taillard. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Annals of Operations* 2005; 139: 65–94.
- [7] J S Kochhar, B T Foster, S S Heragu. HOPE: A genetic algorithm for the unequal area facility layout problem. *Computers & Operations Research* 1998; 25(7-8): 583–594.
- [8] Z Drezner. A new genetic algorithm for the quadratic assignment problem. *Inform Journal on Computing* 2003; 15: 320–330.
- [9] J Skorin-Kapov. Tabu search applied to the quadratic assignment problem, *ORSA Journal on Computing*. 1990; 2: 33–45.
- [10] E Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing* 1991; 17: 443–455.
- [11] A Misevicius, A Osterika. Defining tabu tenure for the quadratic assignment problem. *Information Technology and Control* 2007; 36 (4): 341–347.
- [12] A Misevicius. An implementation of the iterated tabu search algorithm for the quadratic assignment problem. *OR Spectrum* 2012; 34(3): 665-690.
- [13] R Burkard, F Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research* 1984; 17(2): 169–174.
- [14] M Wilhelm, T Ward. Solving quadratic assignment problems by simulated annealing. *IIE Transactions* 1987; 19: 107–119.
- [15] T Abreu, N Querido, N Boaventura. A simulated annealing for the quadratic assignment problem. *Rairo-Operations Research* 1999; 33: 249–273.
- [16] T Stützle and M Dorigo. ACO algorithms for the quadratic assignment problem. In: *New ideas in optimization*, David Corne, Marco Dorigo, Fred Glover, Dipankar Dasgupta, Pablo Moscato, Riccardo Poli, and Kenneth V. Price editors. Maidenhead, UK, England: McGraw-Hill Ltd, 1999. pp. 33-50.
- [17] M Dorigo. The ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B: Cybernetics* 1996; 26 (1): 1–13.

- [18] S Tsutsui, N Fujimoto. Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO 2009), pp. 2523–2530.
- [19] S Tsutsui, N Fujimoto. Fast QAP solving by ACO with 2-opt local search on a GPU. In 2011 IEEE Congress of Evolutionary Computation (CEC 2011). pp. 812–819.
- [20] T Luong, N Melab, E. Talbi. Parallel hybrid evolutionary algorithms on GPU. In IEEE World Congress on Computational Intelligence, 2010.
- [21] W Zhu, J Curry, A Marquez. SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. *International Journal of Production Research* 2010; 48(4): 1035-1047.
- [22] M Czapinski. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform, *Journal of Parallel and Distributed Computing* 2013; 73: 1461–1468.
- [23] J M Cecilia, J M Garcia, A Nisbet, M Amos, M Ujaldon. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing* 2013; 73 (1): 42–51.
- [24] A Delevacq, P Delisle, M Gravel, M Krajecki. Parallel ant colony optimization on graphics processing units, *Journal of Parallel and Distributed Computing* 2013; 73: 52–61.
- [25] C Schulz. Efficient local search on the GPU investigation on the vehicle routing problem. *Journal of Parallel and Distributed Computing* 2013; 73: 14–31.
- [26] C Groer, B Golden, E Wasil. A parallel algorithm for the vehicle routing problem, *Inform Journal on Computing* 2011; 23: 315–330.
- [27] C -S. Huang, Y-C. Huang, P.-J. Lai. Modified genetic algorithms for solving fuzzy flow shop scheduling problems and their implementation with CUDA. *Expert Systems with Applications* 2012; 39: 4999–5005.
- [28] M Czapinski, S Barnes, Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform, *Journal of Parallel and Distributed Computing* 2011; 71: 802–811.
- [29] W Bozejko. On single-walk parallelization of the job shop problem solving algorithms, *Computers & Operations Research* 2012; 39: 2258–2264.
- [30] R T Kneusel. Curve-Fitting on Graphics Processors Using Particle Swarm Optimization, *International Journal of Computational Intelligence Systems*. 2013; 7 (2): 213–224.
- [31] D Kirk, W.-M. W. Whu. Programming massively parallel processors. Burlington, USA: Elsevier Inc., 2010.
- [32] NVIDIA. NVIDIA's next generation CUDA computer architecture Fermi (white paper). NVIDIA, 2011.
- [33] J H Holland. Adaptation in natural and artificial systems. USA: A Bradford Book, 1992.
- [34] R Burkard, S Karisch, F Rendl. QAPLIB-a quadratic assignment problem library. *European Journal of Operational Research* 1991; 55(1): 115–119.