OpenCL implementation of PSO algorithm for the Quadratic Assignment Problem *

Piotr Szwed and Wojciech Chmiel and Piotr Kadłuczka

AGH University of Science and Technology
{pszwed,wch,pkad}@agh.edu.pl

Abstract. This paper presents a Particle Swarm Optimization (PSO) algorithm for the Quadratic Assignment Problem (QAP) implemented on OpenCL platform. Motivations to our work were twofold: firstly we wanted to develop a dedicated algorithm to solve the QAP showing both time and optimization performance, secondly we planned to check, if the capabilities offered by popular GPUs can be exploited to accelerate hard optimization tasks requiring high computational power. We were specifically targeting low-cost popular devices, with limited capabilities. The paper discusses the algorithm and its parallel implementation, as well as reports results of tests.

Keywords: QAP, PSO, OpenCL, GPU calculation, particle swarm optimization, discrete optimization

1 Introduction

Quadratic Assignment Problem (QAP) is considered one of the most fundamental optimization problems, as it generalizes a large number of theoretical issues, including graph partitioning, finding maximal clique or linear arrangement. The QAP can be used to model several practical problems, such as balancing of jet turbines, less-than-truckload (*LTL*), very-large-scale integration (*VLSI*), backboard wiring problem and molecular fitting.

The basic QAP formulation is the following: given a set of n *facilities* and n *locations*, the goal is to find an assignment of facilities to locations that minimizes the goal function, which is calculated as a sum of flows between facilities multiplied by distances between locations. A there are $n!$ possible assignments, the QAP is one of the most difficult combinatorial problems belonging to the *NP-hard* class. Therefore, only approximation algorithms can be used for the case, where the n is bigger than 30 ([1], [2], [3]).

Particle Swarm Optimization (PSO) is an optimization method inspired by an observation of social behavior. It attempts to find an optimal problem solution by moving a population of particles in a search space. Each particle is

* This is a draft version of the paper submited to **Artificial Intelligence and Soft Computing - 13th International Conference, ICAISC 2014**, Zakopane, Poland, June 14-18, 2014. http://icaisc.eu/

characterized by two features its position and velocity. Depending on a method variation, particles may exchange information on their positions and reached values of goal functions [4]. PSO is a metaheuristics, that can be mapped on various domains. Although the method was intended for continuous domains, its applications to discrete problems, including the Traveling Salesman Problem (TSP) and the QAP were discussed in [5–7].

In this paper we present an implementation of PSO algorithm for the QAP problem on OpenCL platform. OpenCL is a solution allowing developers to accelerate applications by using computational power of multicore graphic cards and processors. OpenCL enabled devices are quite widespread, even if often their users don't fully realize it. They include popular components (graphic cards and CPUs) from AMD, Nvidia and Intel companies.

A motivation to our work was to check, if the capabilities offered by popular GPUs can be exploited to accelerate hard optimization tasks requiring high computational power. In this paper we make the following two contributions: firstly we present a developed PSO algorithm for the QAP problem, secondly we discuss its parallel implementation on OpenCL platform.

The paper is organized as follows: next Section 2 gives the definition of QAP. It is followed by Section 3, which discusses the application of PSO to the QAP, as well as its parallel implementation with OpenCL. Experiments performed and their results are presented in Section 4. Section 5 provides concluding remarks.

2 Quadratic AssignmentProblem

Quadratic Assignment Problem was introduced by Koopmans and Beckman in 1957 as a mathematical model of assigning a set of economic activities to a set of locations.

For the given set $N = \{1, ..., n\}$ we define two $n \times n$ non-negative matrices $F = [f_{i,k}], D = [d_{j,l}]$. In the terminology of facilities-location the set N is a set of facilities indexes and $\pi(i) \in N, i = 1, ..., n$ defines locations, to which the facilities are assigned. Matrix D defines distances between locations, whereas matrix F defines flows between pairs of facilities. Matrix B describes a linear part of the assignment cost and in most cases is omitted. A solution of QAP (also denoted as $QAP(F, D)$) can be defined as a permutation $\pi = (\pi(1), ..., \pi(n))$ from the set of n facilities. In the Koopman-Beckman's [8] model the goal is to find the permutation π^* which minimizes the objective function:

$$f(\pi^*) = min_{\pi \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\pi(i),\pi(j)} + \sum_{i=1}^{n} b_{i,\pi(i)} \tag{1}$$

The objective function $f(\pi), \pi \in \Pi$ describes the global cost of system realization and exploitation. Π is a set of permutations of the set of natural numbers $1, \ldots, n$. In most cases matrix D and F are symmetric: distances $d_{i,j}$ and $d_{j,i}$ between two locations i and j are equal, the same applies to flows: $f_{i,j}$ and $f_{j,i}$.

QAP models found application in various areas including transportation [9], scheduling, electronics (wiring problem), distributed computing, statistical data

analysis (reconstruction of destroyed soundtracks), balancing of turbine running [10], chemistry [11], genetics [12], creating the control panels and manufacturing [13].

In 1976 Sahni and Gonzalez proved that the QAP is strongly $\mathcal{NP}$-*hard* [14, 15], by showing that a hypothetical existence of a polynomial time algorithm for solving the QAP would imply an existence of a polynomial time algorithm for an $\mathcal{NP}$-*complete* decision problem - the Hamiltonian cycle.

In many cases finding an optimal solution for the QAP by applying local search is very hard. The neighborhood definition often used in algorithms solving the QAP is the structure *2-opt* (based on a pair exchange in a permutation). Fig. 1 shows an example of landscape for the problem instance *Lipa60b*. As it could be seen, this landscape (*QAP, 2-opt*) is multimodal. The neighborhood solutions are characterized by weak autocorrelation, hence, this instance of QAP (and many others) is difficult to optimize. Several approximation algorithms for the QAP use procedures based on local search, but on the basis the above considerations, it can be proven that in a general case this approach does not guarantee finding a good solution.

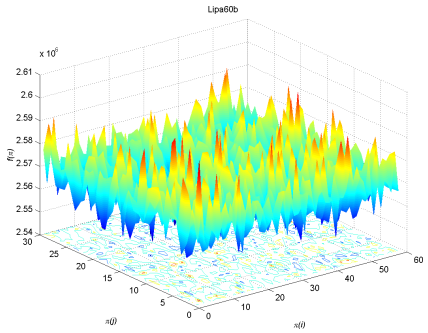

Fig. 1. An example of the landscape for the QAP problem (Lipa60) for *2-opt* neighborhood structure.

3 Methods

The classical PSO algorithm [4] is an optimization method defined for continuous domain. During the optimization process a number of particles move through a search space and update their states and values of goal function at discrete time steps $t = 1, 2, 3, \ldots$ Each particle is characterized by its position $x(t)$ and velocity $v(t)$. A particle remembers its best position reached so far $p^L(t)$, as well as it can use information about the best solution found by the swarm $p^G(t)$.

The state equation for a particle is given by formula (2). Coefficients $c_1, c_2, c_3 \in [0, 1]$ are called respectively *inertia*, *cognition* (or *self recognition*) and *social* factors.

$$\left.\begin{aligned} v(t+1) &= c_1 \cdot v(t) + c_2 \cdot (p^L(t) - x(t)) + c_3 \cdot (p^G(t) - x(t)) \\ x(t+1) &= x(t) + v(t) \end{aligned}\right\} \quad (2)$$

An adaptation of the PSO method to a discrete domain necessities in giving interpretation to the velocity concept, as well as defining equivalents of scalar multiplication, subtraction and addition for arguments being solutions and velocities. Examples of such interpretations can be found in [5] for the TSP and [6] for the QAP.

In the rest of this section we describe an adaptation of the Particle Swarm Optimization (PSO) method to the QAP problem. Some solutions, especially the interpretation of the velocity, are based ideas presented in [7].

3.1 PSO adaptation for the QAP problem

A state of a particle is a pair (X, V). For the QAP problem both are $n \times n$ matrices, where n is the problem size. The matrix $X = [x_{ij}]$ encodes an assignment of facilities to locations. Its elements x_{ij} are equal to 1, if j-th facility is assigned to i-th location, and take value 0 otherwise.

A particle moves in the solution space following the direction given by the velocity V. Elements v_{ij} have the following interpretation: if v_{ij} has high positive value, then a procedure determining the next solution should favor an assignment $x_{ij} = 1$. On the other hand, if $v_{ij} \leq 0$, then $x_{ij} = 0$ should be preferred.

The state of a particle reached in t-th iteration will be denoted by $(X(t), V(t))$. In each iteration a state of a particle is updated according to formulas (3) and (4).

$$V(t+1) = S_v(c_1 \cdot V(t) + c_2 \cdot r_2(t) \cdot (P^L(t) - X(t)) + c_2 \cdot r_3(t) \cdot (P^G(t) - X(t))) \quad (3)$$

$$X(t+1) = S_x(X(t) + V(t)) \quad (4)$$

Coefficients r_2 and r_3 are random numbers from $[0, 1]$ generated for each particle and iteration. They are introduced to model a random choice between movements in the previous direction (according to c_1 – inertia), the best local solution (self recognition) or the global best solution (social behavior).

All operators appearing in (3) and (4) are standard operators from linear algebra. Instead of redefining them for a particular problem, see e.g. [5], we propose to use aggregation functions S_v and S_x that allow to adapt the algorithm to particular needs of a discrete problem.

The function S_v is used to assure that particle velocity have reasonable values. Initially, we thought that unconstrained growth of velocity can be a problem, therefore we have implemented a function, which restricts the elements of V to an interval $[-v_{max}, v_{max}]$. This function is referred as *raw* in Table 2. However, the experiments conducted shown, that in case of small inertia factor, e.g. $c_1 = 0.5$,

after a few iterations all velocities tend to 0 and in consequence all particles converge to the best solution encountered earlier by the swarm. To avoid such effect another function was applied, which additionally performs column normalization. For each column j a sum of absolute values of the elements $n_j = \sum_{i=1}^{n} |v_{ij}|$ is calculated and then the following assignment is made: $v_{ij} \leftarrow v_{ij}/n_j$.

According to formula (4) a new particle position $X(t+1)$ is obtained by aggregating the previous state components: $X(t)$ and $V(t)$. As elements of a matrix $X(t) + V(t)$ may take values from $[-v_{max}, v_{max} + 1]$, the S_x function is responsible for converting it into a valid assignment matrix having exactly one 1 in each row and column. Actually, S_v is rather a procedure, than a function, as it incorporates some elements of random choice.

Three variants of S_x procedures were implemented:

1. $GlobalMax(X)$ – iteratively searches for x_{rc}, a maximum element in a matrix X, sets it to 1 and clears other elements in the row r and c.
2. $PickColumn(X)$ – picks a column c from X, selects a maximum element x_{rc}, replaces it by 1 and clears other elements in r and c.
3. $SecondTarget(X)$ – similar to $GlobalMax(X)$, discussed in detail in section 3.2.

Due to limited space we present only the algorithm for $GlobalMax$ (Algorithm 1). In a **while** loop, executed exactly n times, it calculates M, the set of maximum elements in the input matrix $X(t) + V(t)$, whose row and column indices belong to the sets R and C respectively. Then, it picks an element x_{rc} from M (if it has more then one elements), clears elements in the row r and the column c and sets x_{rc} to 1. Hence, the selected assignment represents the best choice, considering previous decisions (which in some cases can be random). Initially, R and C contain all indices $1, \ldots, n$. In each iteration exactly one (r or c) is removed from both sets, hence the procedure stops after n iterations.

3.2 Second target aggregation procedure

In several experiments, where which $GlobalMax$ aggregation procedure was used, particles seemed to get stuck, even if their velocities were far from zero. We reproduce this effect on a small 3×3 example:

$$X = \begin{bmatrix} 1\,0\,0 \\ 0\,0\,1 \\ 0\,1\,0 \end{bmatrix} V = \begin{bmatrix} 7\,1\,3 \\ 0\,4\,5 \\ 2\,3\,2 \end{bmatrix} X + V = \begin{bmatrix} 8\,1\,3 \\ 0\,4\,6 \\ 2\,4\,2 \end{bmatrix} S_x(X+V) = \begin{bmatrix} 1\,0\,0 \\ 0\,0\,1 \\ 0\,1\,0 \end{bmatrix}$$

For the described case in subsequent iterations it will hold $X(t+1) = X(t)$, until another particle is capable of changing $(P^G(t) - X(t)))$ component of formula (3) for velocity calculation. A solution for this problem can be to move a particle to a secondary direction, by ignoring $k < n$ elements that are in the solution $X(t)$ already set to 1. This, depending on k, gives an opportunity to reach other solutions with a smaller goal function value (see Fig. 2). If they are maximum elements in the remaining matrix denoted here as $X \oslash_k V$, they are

Algorithm 1 Aggregation procedure $GlobalMax$)

```
1: procedure GlobalMax(X)
2:     R ← {1, ..., n}
3:     C ← {1, ..., n}
4:     while R ≠ ∅ ∧ C ≠ ∅ do
5:         M ← {(r, c): ∀_{i∈R,j∈C}(x_rc ≥ x_ij)}    ▷ Calculate set of maximum elements
6:         Randomly select (r, c) from M
7:         R ← R \ {r}                                ▷ Update the sets R and C
8:         C ← C \ {c}
9:         for i in [1, n] do
10:            x_ri ← 0                               ▷ Clear r-th row
11:            x_ic ← 0                               ▷ Clear c-th column
12:        end for
13:        x_rc ← 1                                   ▷ Assign for 1 for a maximum value
14:    end while
15:    return X
16: end procedure
```

still reasonable movement directions. Formula (5) shows $X \oslash_k V$ matrix for $k = 3$ in the discussed example. Elements of a new solution are marked with circles.

$$X \oslash_{k=3} V = \begin{bmatrix} 0 & 1 & \textcircled{3} \\ 0 & \textcircled{4} & 0 \\ \textcircled{2} & 0 & 2 \end{bmatrix} \tag{5}$$

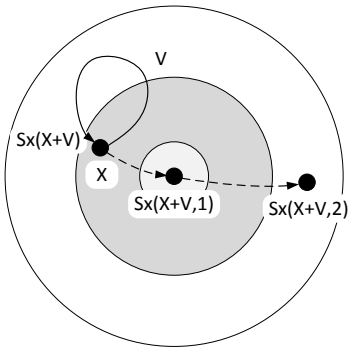

Fig. 2. An idea of the *second target* aggregation function

3.3 OpenCL platform

OpenCL [16] is a standard providing a common language, programming interfaces and hardware abstraction for heterogeneous platforms including GPU, mul-

ticore CPU, DSP and FPGA [17]. It allows to accelerate computations by decomposing them into a set of parallel tasks (work items) operating on separate data.

A program on OpenCL platform is decomposed into two parts: sequential executed by the CPU *host* and parallel executed by multicore *devices*. Functions executed on devices are called *kernels*. They are written in a language being a variant of C with some restrictions related to keywords and datatypes. When first time loaded, the kernels are automatically translated into a target device instruction set. The whole process takes about 500ms.

OpenCL supports 1D, 2D or 3D organization of data (arrays, matrices and volumes). Each data element is identified by 1 to 3 indices, e.g. $d[i][j]$ for two-dimensional arrays. A *work item* is a scheduled kernel instance, which obtain a combination of data indices within the data range. To give an example, a 2D array of data of $n \times m$ size should be processed by $n \cdot m$ kernel instances, which are assigned with a pair of indices (i, j), $i < n$ and $j < m$. Those indices are used to identify data items assigned to kernels.

Additionally, kernels can be organized into workgroups, e.g. corresponding to parts of a matrix, and synchronize their operations within a group using so called *local barrier* mechanism. However, workgroups suffer from several platform restrictions related to number of work items and amount of accessible memory.

OpenCL uses three types of memory: global (that is exchanged between the host and the device), local for a work group and private for a work item.

3.4 OpenCL algorithm implementation

In our implementation we used *aparapi* platform [18] that allows to write OpenCL programs directly in Java language. The platform comprises two parts: an API and a runtime capable of converting Java bytecodes into OpenCL workloads. Hence, the host part of the program is executed on a Java virtual machine, and originally written in Java kernels are executed on an OpenCL enabled device.

The basic functional blocks of the algorithm are presented in Fig. 3. Implemented kernels are marked with gray color. The code responsible for generation of random particles is executed by the host. We have also decided to leave the code for updating best solutions at the host side. Actually, it comprises a number of native `System.arraycopy()` calls.

Each particle is represented by a number of matrices (see Fig. 4): X and X_{new} – solutions, P^L – local best particle solution and V – velocity. Moreover, particles share read-only global best solution – P^G and generated by the host arrays of random numbers (there is no `rand()` equivalent on OpenCL platform). The amount of the memory used can be high. It can be estimated, that for the biggest test case reported in Table 1: 10000 particles using 60×60 matrices, the global memory GPU consumption ranged at 550MB.

An important decision related to OpenCL program design is related to data ranges selection. The memory layout in Fig. 4 suggests 3D range, whose dimensions are: row, column and particle number. This can be applied for relatively simple velocity or goal function calculation. However, the proposed algorithms

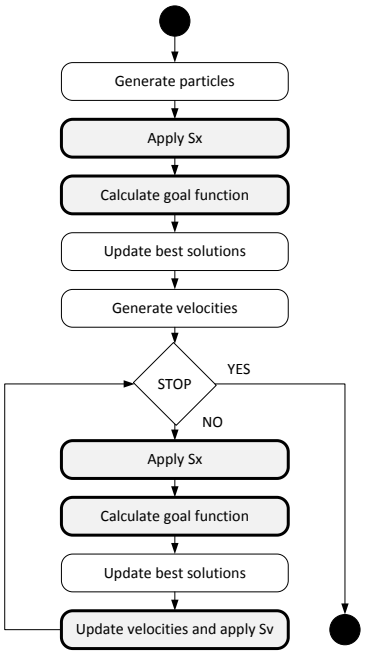

Fig. 3. Functional blocks of OpenCL based algorithm implementation.

for S_x, see Algorithm 1, are far too complicated to be implemented as a simple parallel work item. In consequence, we decided to use only one dimension representing a particle number, what implicates that parallel work items process whole particles. To give an example, X components being 60×60 matrices of all 100 particles are represented by a single array of 360000 floats with a range $i = 0, \ldots, 99$.

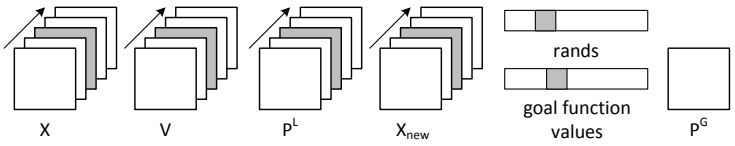

Fig. 4. Global variables used in the algorithm implementation

4 Experiments and results

We have conducted two types of experiments. The first aimed at evaluating the time performance of GPU based implementation for various setups of PSO

algoritms (varying numbers of particles and numbers of iterations). The goal of the second group of test cases was to establish the influence of parameters controlling the implemented PSO algorithm on its optimization efficiency. All tests were performed on instances defined in QAPLIB problem library [19].

4.1 Time performance

The OpencCL implementation was tested on two platforms, referred as *laptop* (AMD Radeon HD 6750M card, i7-2657QM, 2.2Ghz processor, Windows 7) and *workstation* (NVIDIA GeForce GT 430, i7-4860HQ processor, 3.60GHz, Windows 7)). In both cases Java 8 runtime was used for host operations.

Fig. 5 gives the average times spent in one iteration for various numbers of particles. It should be noted, that at the workstation platform it was not possible to run tests for the problem size 100.

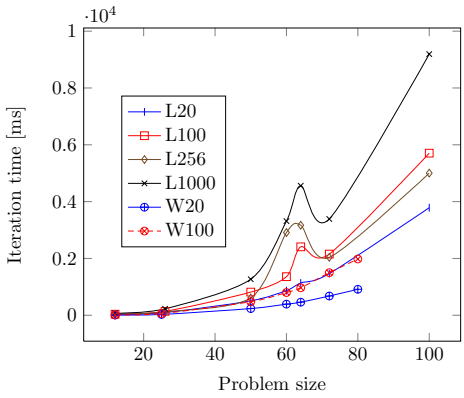

Fig. 5. Time spent in one iteration for various problem sizes. L20, L100, L256 and L1000: 20, 100, 256 and 1000 particles (laptop), W20 and W100: 20 and 100 particles (workstation)

Detailed results of tests related to execution time are given in Table 1. It can be observed, that the tested parallel implementation is inferior to the sequential, if the number of particles is relatively small. For 20 or even 100 particles the overhead related to data transfer between the host and the GPU prevails potential benefits.

A real speedup can be observed for 200 or more particles being simultaneously processed. This is visible in last eight table rows giving results for 1000 and 10000 particles. The results suggest quite different algorithm design, e.g. to exploit the platform capabilities sequential algorithm runs for 20 particles should be transformed into independent 500 parallel runs.

It should be noted, however, that all tests were not performed on a dedicated GPU hardware, but on popular graphic cards installed in mid-range laptops or workstations.

Table 1. Comparison of iteration times for parallel and sequential implementations. All times (PAR and SEQ) expressed in ms.

Problem size	Particles	Iterations	Time PAR [ms]	Sx	Goal	Best	Velocity	Time SEQ [ms]	Gain: $\frac{\text{SEQ}}{\text{PAR}}$
12	20	100	1.42	63.53%	17.40%	0.19%	18.89%	0.07	0.05
26	20	100	4.75	69.98%	19.62%	0.47%	9.93%	0.55	0.12
50	20	100	25.40	76.90%	19.57%	0.31%	3.23%	3.61	0.14
60	20	100	43.46	76.37%	21.10%	0.06%	2.46%	5.16	0.12
64	20	100	56.80	57.70%	38.33%	0.02%	3.94%	5.57	0.10
72	20	100	73.35	76.23%	21.60%	0.20%	1.97%	6.21	0.08
100	20	100	189.29	75.92%	22.60%	0.16%	1.32%	9.84	0.05
12	100	100	0.38	63.53%	17.40%	0.19%	18.89%	0.11	0.30
26	100	100	1.38	69.98%	19.62%	0.47%	9.93%	0.79	0.57
50	100	100	8.15	76.90%	19.57%	0.31%	3.23%	4.69	0.58
60	100	100	13.59	76.37%	21.10%	0.06%	2.46%	6.76	0.50
64	100	100	24.08	57.70%	38.33%	0.02%	3.94%	9.64	0.40
72	100	100	21.54	76.23%	21.60%	0.20%	1.97%	8.15	0.38
100	100	100	57.06	75.92%	22.60%	0.16%	1.32%	12.96	0.23
12	1000	10	0.07	47.42%	23.90%	0.44%	28.24%	0.15	2.12
26	1000	10	0.23	53.33%	27.41%	2.76%	16.50%	1.04	4.56
50	1000	10	1.27	53.75%	37.92%	1.51%	6.82%	6.71	5.29
60	1000	10	3.31	72.54%	23.12%	0.56%	3.78%	7.12	2.15
64	1000	10	4.56	44.95%	41.54%	0.30%	13.20%	12.38	2.72
72	1000	10	3.39	57.58%	36.68%	1.02%	4.72%	10.79	3.18
100	1000	10	9.19	56.17%	40.36%	0.38%	3.09%	17.51	1.91
60	10000	100	1.61	42.38%	47.74%	0.02%	9.86%	12.18	7.56

4.2 Optimization performance

The second group of tests aimed at establishing the optimization performance of the algorithm for various combinations of parameters (including kernels used). The tests were performed a randomly generated problem *Tai60b* from the QAPLIB collection [19]. The best known goal function value (608215054) for *Tai60b* was established with a robust Tabu search algorithm [20]. We consider it a reference in the comparisons.

Table 2 gives selected results of tests, which yielded the bests, average and the worst results. It can be observed that the best solutions were obtained for large numbers of iterations (the reference value for *Tai60b* was also obtained the number of iterations in order of 100000 [20]). In most cases *raw* S_v function (without normalization) returned worse results than *Norm*. For S_x aggregation function, results of applying global maximum (*GMax*) and pick column (*PCol*) are comparable. The kernel implementing the second target (*STarget*) gave the best results. Good results were reported for equal values of c_1, c_2 and c_3 coefficients. It may be stated that c_2 (self recognition) should not dominate other factors, whereas high inertia c_1 is acceptable.

Table 2. Results of multiple tests for Tai60b (problem size: 60)

Particles	Iterations	Inertia c_1	Self recognition c_2	Social factor c_3	Velocity kernel	Sx kernel	STarg depth	First goal	Reached goal	Gain	Iteration	Gap
100	3000	0.5	0.5	0.5	Norm	STarg	0.25	903886656	659155648	27.08%	2840	7.73%
100	6000	0.5	0.5	0.5	Raw	STarg	0.25	903886656	661662080	26.80%	5852	8.08%
100	3000	0.9	0.3	0.3	Norm	STarg	0.75	903886656	664190656	26.52%	2715	8.43%
50	1000	0.5	0.5	0.5	Raw	STarg	0.25	932608128	666681280	28.51%	904	8.77%
1000	6000	0.5	0.5	0.5	Norm	STarg	0.25	899822016	669809536	25.56%	5755	9.20%
500	100	0.5	0.5	0.5	Raw	STarg	0.5	901977728	719012736	20.28%	50	15.41%
200	250	0.9	0.3	0.3	Raw	STarg	0.5	903886656	731279680	19.10%	176	16.83%
500	100	0.3	0.9	0.3	Raw	STarg	0.75	901977728	734171136	18.60%	99	17.16%
200	250	0.9	0.3	0.3	Raw	PCol	N/A	902798080	736113152	18.46%	67	17.37%
500	100	0.5	0.5	0.5	Raw	STarg	0.5	901977728	741909632	17.75%	100	18.02%
50	1000	0.5	0.5	0.5	Raw	GMx	N/A	902798080	858625408	4.89%	11	29.16%
500	100	0.9	0.3	0.3	Raw	GMx	N/A	901977728	871849536	3.34%	72	30.24%
500	100	0.5	0.5	0.5	Raw	GMx	N/A	901977728	877890624	2.67%	83	30.72%
500	100	0.3	0.9	0.3	Raw	PCol	N/A	890460224	879615744	1.22%	3	30.85%
50	1000	0.3	0.9	0.3	Raw	PCol	N/A	902798080	902798080	0.00%	0	32.63%
200	250	0.3	0.9	0.3	Raw	PCol	N/A	902798080	902798080	0.00%	0	32.63%

5 Conclusions

In this paper we describe a PSO algorithm designed for solving the QAP problem, as well as its parallel implementation on the OpenCL platform. Several mechanisms applied in the algorithm were inspired by Liu at al. work [7], however, they were refined to provide better performance.

Another contribution of this work is a parallel implementation of the discussed algorithm on the OpenCL platform. We developed a Java program that uses *aparapi* library to deliver computational tasks to an OpenCL enabled device. We were specifically targeting low-cost popular devices, e.g. 200$ graphics cards, with limited capabilities.

We report results of tests aiming at evaluating the implementation in terms of execution times and optimization capability. The tests targeting time performance revealed that benefits of GPU calculations can be observed, if the number of particles processed in parallel is big. The optimization performance, here presented on a selected large QAP instance ($n = 60$), showed that the algorithm behaves differently, depending on values of control parameters. However, the proposed *second target* method for updating particle position yielded better results, than the others.

References

1. Burkard, R., Karisch, S., Rendl, F.: Qaplib-a quadratic assignment problem library. Journal of Global Optimization **10**(4) (1997) 391–403

2. Chmiel, W.: Evolution Algorithms for optimisation of task assignment problem with quadratic cost function. PhD thesis, AGH Technology University, Poland (2004)
3. Chmiel, W., Kadłuczka, P., Packanik, G.: Performance of swarm algorithms for permutation problems. Automatyka **15**(2) (2009) 117–126
4. Eberhart, R., Kennedy, J.: A new optimizer using particle swarm theory. In: Micro Machine and Human Science, 1995. MHS '95., Proceedings of the Sixth International Symposium on. (Oct 1995) 39–43
5. Clerc, M.: Discrete particle swarm optimization, illustrated by the traveling salesman problem. In: New optimization techniques in engineering. Springer (2004) 219–239
6. Onwubolu, G.C., Sharma, A.: Particle swarm optimization for the assignment of facilities to locations. In: New Optimization Techniques in Engineering. Springer (2004) 567–584
7. Liu, H., Abraham, A., Zhang, J.: A particle swarm approach to quadratic assignment problems. In Saad, A., Dahal, K., Sarfraz, M., Roy, R., eds.: Soft Computing in Industrial Applications. Volume 39 of Advances in Soft Computing. Springer Berlin Heidelberg (2007) 213–222
8. Koopmans, T.C., Beckmann, M.J.: Assignment problems and the location of economic activities. Econometrica **25** (1957) 53–76
9. Bermudez, R., Cole, M.H.: A genetic algorithm approach to door assignments in breakbulk terminals. Technical Report MBTC-1102, Mack-Blackwell Transportation Center, University of Arkansas, Fayetteville, Arkansas (2001)
10. Mason, A., Rönnqvist, M.: Solution methods for the balancing of jet turbines. Computers & OR **24**(2) (1997) 153–167
11. Ugi, I., Bauer, J., Brandt, J., Friedrich, J., Gasteiger, J., Jochum, C., Schubert, W.: Neue anwendungsgebiete fur computer in der chemie. Angewandte Chemie **91**(2) (1979) 99–111
12. Phillips, A.T., Rosen, J.B.: A quadratic assignment formulation of the molecular conformation problem. JOURNAL OF GLOBAL OPTIMIZATION **4** (1994) 229–241
13. Grötschel, M.: Discrete mathematics in manufacturing. In Malley, R.E.O., ed.: ICIAM 1991: Proceedings of the Second International Conference on Industrial and Applied Mathematics, SIAM (1991) 119–145
14. Sahni, S., Gonzalez, T.: P-complete approximation problems. J. ACM **23**(3) (1976) 555–565
15. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
16. Howes, L., Munshi, A.: The OpenCL specification. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf Online: last accessed: Jan 2015.
17. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering **12**(3) (2010) 66
18. Howes, L., Munshi, A.: Aparapi - AMD. http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/ Online: last accessed: Jan 2015.
19. Burkard, R., Karisch, S., Rendl, F.: QAPLIB a Quadratic Assignment Problem library. Journal of Global Optimization **10**(4) (1997) 391–403
20. Taillard, E.D.: Comparison of iterative searches for the quadratic assignment problem. Location Science **3**(2) (1995) 87 – 105