Parallelization Strategies for Hybrid Metaheuristics using a Single GPU and Multi-core Resources

Thé Van Luong, Eric Taillard, Nouredine Melab, and El-Ghazali Talbi

[1] HEIG-VD, Yverdon-les-Bains, SWITZERLAND,
`The-Van.Luong@heig-vd.ch, Eric.Taillard@heig-vd.ch`
[2] INRIA Lille Nord Europe / LIFL, Villeneuve d'Ascq, FRANCE,
`Nouredine.Melab@lifl.fr, talbi@lifl.fr`

Abstract. Hybrid metaheuristics are powerful methods for solving complex problems in science and industry. Nevertheless, the resolution time remains prohibitive when dealing with large problem instances. As a result, the use of GPU computing has been recognized as a major way to speed up the search process. However, most GPU-accelerated algorithms of the literature do not take benefits of all the available CPU cores. In this paper, we introduce a new guideline for the design and implementation of effective hybrid metaheuristics using heterogeneous resources.

1 Introduction

Metaheuristics are approximate methods that make it possible to solve in a reasonable time NP-hard complex problems. Two main categories are distinguished: population-based metaheuristics (P-metaheuristics) and solution-based metaheuristics (S-metaheuristics). Theoretical and experimental studies have shown that the hybridization between these two classes may improve the quality of provided solutions [1]. However, as it is time-consuming, there is often a compromise between the number of solutions to use and the computational complexity to explore it.

Recently, graphics processing units (GPU) have emerged as a popular support for massively parallel computing [2]. To the best of our knowledge, most GPU-accelerated metaheuristics designed in the literature only exploit a single CPU core. This is typically the case for hybrid metaheuristics on GPU [3–5]. Thus, it might be valuable to fully utilize the other remaining CPU resources. It may be particularly significant when the acceleration factors obtained by the GPU-based algorithm are relatively modest. Indeed, since all processors are nowadays multi-core, performance of GPU-based algorithms might be improved.

Nevertheless, designing optimization methods on such a heterogeneous architecture is not straightforward. Indeed, the major issues are mainly related to the distribution of tasks processing between the GPU and CPU cores. In this paper, we introduce a general guideline to deal with such issues. We propose the re-design of hybrid metaheuristics on GPU taking advantage of every available CPU cores. In this purpose, an efficient distribution of the search process between the GPU and the CPU is done. At the same time, an efficient load balancing between the GPU and the remaining CPU cores is proposed to fully utilize all the available heterogeneous resources.

As an example of application, the quadratic assignment problem (QAP) has been considered. Such a problem provides interesting irregular properties, since for optimized S-metaheuristics, most of move evaluations can be done in constant time. Thereby, speed-ups from a parallel implementation are expected to be relatively modest. Hence, the use of multi-core resources in addition with GPU-based metaheuristics is clearly meaningful.

The remainder of the paper is organized as follows: Section 2 highlights the principles of parallel models for metaheuristics on GPU. In Section 3, parallelization concepts for designing hybrid metaheuristics on GPU are described. An extension of these approaches is investigated in Section 4 for exploiting heterogeneous resources. Section 5 reports the performance results obtained for the QAP. Finally, some conclusions of this work are drawn in Section 6.

2 Parallel Metaheuristics on GPU

2.1 Parallel Models of Metaheuristics

In general, for hybrid metaheuristics, executing the iterative process of a S-metaheuristic (e.g. a local search) requires a large amount of computational resources. Consequently, parallelism arises naturally when dealing with a neighborhood. In this purpose, three major parallel models for metaheuristics can be distinguished [6]: solution-level, iteration-level and algorithmic-level.

- *Solution-level Parallel Model.* The focus is on the parallel evaluation of a single solution. Problem-dependent operations performed on solutions are parallelized. That model is particularly interesting when the evaluation function can be itself parallelized, as it is CPU time-consuming and/or IO intensive.
- *Iteration-level Parallel Model.* This model is a low-level Master-Worker model that does not alter the behavior of the heuristic. The evaluation of solutions is performed in parallel. An efficient execution is often obtained especially when the evaluation of each solution is costly.
- *Algorithmic-level Parallel Model.* Several metaheuristics are simultaneously launched for computing better and robust solutions. They may be heterogeneous or homogeneous, independent or cooperative, start from the same or different solution(s), configured with the same or different parameters.

2.2 Metaheuristics on GPU Architectures

Recently, GPU accelerators have emerged as a powerful support for massively parallel computing. Indeed, these architectures offer a substantial computational horsepower and a high memory bandwidth compared to CPU-based architectures. Due to their inherent parallel nature, P-metaheuristics such as evolutionary algorithms have been the first subject of parallelization on GPU: genetic algorithms [7], particle swarm optimization [8], ant colonies [9] and so on.

Regarding S-metaheuristics, the parallelization on GPU architectures is much harder, due to the improvement of a single solution. Therefore, only few research works have been investigated for local search algorithms [10–12]. The same goes on when dealing with hybrid metaheuristics on GPU, where there exists only few parallelization approaches [3–5].

3 Design of Parallel Hybrid Metaheuristics on GPU

3.1 Parallel Evaluation of Solutions on GPU

The parallel iteration-level model has to be designed according to the data-parallel single program multiple data model of GPUs. The CPU-GPU task partitioning is such that the CPU executes the entire sequential part of the handled metaheuristic. The GPU is in charge of the evaluation of the solutions set at each iteration. In this model, a function code called kernel is sent to the GPU to be executed by a large number of threads grouped into blocks.

This parallelization strategy has been widely used for P-metaheuristics on GPU especially for evolutionary algorithms due to their intrinsic parallel workload (e.g. in [7]). One of the major issues is to optimize the data transfer between the CPU and the GPU. Indeed, the GPU has its own memory and processing elements that are separate from the host computer.

When it comes to parallelization, the optimization of data transfers is more prominent for S-metaheuristics. As a result, when designing hybrid metaheuristics, the focus is on the embedded S-metaheuristic. In this purpose, we have contributed in [13] for the parallel evaluation of solutions (iteration-level) for local search algorithms. The key point of this approach is to generate the neighborhood of the S-metaheuristic at hand on the GPU side. Such a parallelization strategy makes it possible to minimize data transfers through the PCIe bus: the solution which generates the neighborhood and the resulting fitnesses (see Figure 1).

3.2 Parallelization Strategies for Hybrid Metaheuristics

The previous parallelization approach stands for one S-metaheuristic on GPU according to the iteration-level. For designing GPU-accelerated hybrid metaheuristics that involve a population of solutions, the algorithmic-level parallel model has to be deeply examined. In other words, multiple executions of S-metaheuristics on GPU have to be considered. For achieving this, previous approaches from the iteration-level must be adapted for the algorithmic-level. In this purpose, there are fundamentally two parallelization strategies:

- *One neighborhood evaluation on GPU.* This approach consists in evaluating one neighborhood (a set of solutions) at a time on GPU. According to Figure 1, a possible interpretation could be to repeat the whole process (i.e. the repetition of the execution of a single S-metaheuristic on GPU) to deal with as many S-metaheuristics as needed. The drawback of this approach is that the number of threads executed for one kernel on GPU might not be

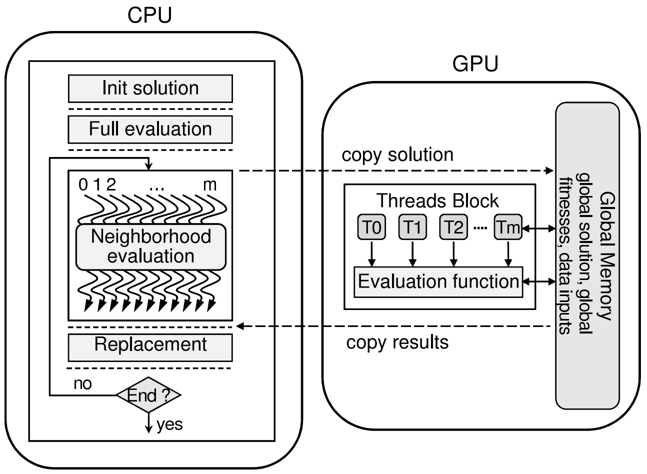

Fig. 1: For S-metaheuristics, the generation and the evaluation of the neighborhood is performed on GPU, and the CPU executes the sequential part of the search process.

enough to cover the memory access latency for few optimization problems. As a result, in the rest of the paper, we will not consider this approach.

- *Many neighborhood evaluations on GPU.* In the second approach, many neighborhoods are evaluated at a time on GPU. For instance, given a certain iteration, if k embedded S-metaheuristics have to be performed on k solutions, the k associated neighborhoods will be generated and evaluated on GPU at the same time. Regarding the thread organization, a thread is associated with many neighbor calculations. For example, a thread block might represent a particular neighborhood from a given S-metaheuristic. Such a parallelization strategy deals with the issues encountered in the first approach since 1) there are enough calculations to keep the GPU multiprocessors busy; 2) the creation overhead of multiple kernel calls is reduced. However, in this second approach, homogeneous embedded S-metaheuristics are required. In such a case, the semantic of the original sequential algorithm might be altered.

4 Parallelization Strategies for Heterogenous Resources

4.1 Multiple S-metaheuristics on Multi-core Architectures

Parallelization approaches for hybrid metaheuristics on GPU presented in the previous section only exploit one single CPU core. To exploit the remaining computational capabilities, thread-based approaches on CPU have to be examined.

In general, for a hybrid metaheuristic, a certain number of independent tasks is likely to be performed in parallel (e.g. a number of S-metaheuristic executions). Therefore, the algorithmic-level parallel model is particularly adapted to CPU architectures, since processes distributed among CPU threads do not necessary share the same instructions and the same execution context.

Algorithm 1 Template for each thread on multi-core CPUs

```
Require: p tasks and number_threads;
1: offset := p / number_threads;
2: tid := get_thread_id();
3: for k = tid * offset; k < tid * offset + offset; k++ do
4:     S-metaheuristic(k);
5: end for
6: if tid < (p mod number_threads) then
7:     k = offset * number_threads + tid;
8:     S-metaheuristic(k);
9: end if
```

Algorithm 1 provides a template for processing independent S-metaheuristics on multi-core architectures. Basically, p tasks (i.e. p S-metaheuristics) have to be equally distributed among the different threads. Each CPU thread is in charge of executing a specific number of S-metaheuristics (lines 1 and 2). Such a realization is performed in a sequential manner (lines 3 to 5). If the number of tasks is not proportional to the number of available cores, remaining tasks will be assigned to the first CPU threads (lines 6 to 9).

4.2 Hybrid Metaheuristics using Heterogeneous Resources

As previously said, one CPU thread is actually associated with the GPU-based algorithm. The major idea for designing a hybrid metaheuristic is to manage the other CPU threads to overlap the calculations performed on GPU. Nevertheless, most of the time, in hybrid metaheuristics, the search process evolves in a synchronous manner at each iteration.

Algorithm 2 Hybrid metaheuristic template using heterogeneous resources

```
Require: m tasks, p tasks and n cores;
1: repeat
2:     Hybrid metaheuristic pre-treatment on host side
3:     S-metaheuristic_multi-core(p,n-1) overlap
4:     S-metaheuristic_gpu(m)
5:     Join results
6:     Hybrid metaheuristic post-treatment on host side
7: until a stopping criterion satisfied
```

For dealing with this issue, we provide in Algorithm 2 a general template for hybrid metaheuristics using heterogeneous resources. Let k be the number of tasks to assess, m the number assigned to the GPU using one CPU core, and p the number assigned to the remaining CPU cores. As quoted above, p S-metaheuristics are executed in parallel on CPU cores (number of available cores minus one) according to Algorithm 1 (line 3). Parallel techniques must be performed to obtain overlapping calculations. Meanwhile, m S-metaheuristics are evaluated on GPU as described in Section 3 (line 4). Then, a synchronization point is set to gather all the obtained results (line 5). Post-treatment operations on the hybrid metaheuristic can be applied afterwards. The process is repeated until a certain criterion is satisfied.

4.3 Load Balancing Heuristic

The remaining issue is to find an efficient load balancing between 1) the GPU using one single core; 2) the remaining CPU cores. Such a task repartition must be done in accordance with the computational capability of heterogeneous resources. We propose in Algorithm 3 a heuristic for doing this load balancing in an efficient way. The major idea of this heuristic is to automatically tune previous m and p parameters during the first iterations of the hybrid metaheuristic at hand.

Algorithm 3 Template for load balancing heuristic

```
Require: k tasks and n cores;
 1: m := ceil (k / 2); p := floor (k / 2);
 2: repeat
 3:   Hybrid metaheuristic pre-treatment on host side
 4:   gpu_time := time (S-metaheuristic_gpu(m));
 5:   cpu_time := time (S-metaheuristic_multi-core(p,n-1));
 6:   Hybrid metaheuristic post-treatment on host side
 7:   relat_speedup := cpu_time / gpu_ time;
 8:   if relat_speedup > 1 then
 9:     potential_p := p / relat_speedup; mult_coeff := k / (m + potential_p);
10:     m := round (m * mult_coeff);
11:     p := round (potential_p * mult_coeff);
12:   else
13:     potential_m := m * rel_speedup; mult_coeff := k / (p + potential_m);
14:     m := round (potential_m * mult_coeff);
15:     p := round (p * mult_coeff);
16:   end if
17: until a certain number of trials
Ensure: m tasks and p tasks;
```

At the beginning of the algorithm, tasks are equally divided between the GPU and the CPU cores (line 1). Then, the time measurement of m S-metaheuristic executions on GPU using one CPU core is accomplished (line 4). The same goes on for the creation of p S-metaheuristics on the other available CPU cores (line 5). Thereafter, the relative speed-up between the two versions is calculated (line 7). If the time to evaluate m tasks on GPU is less important than the time to compute p tasks on remaining CPU cores, then more tasks will be assigned to the GPU during the next iteration (lines 8 to 11). Otherwise, more tasks will be assigned to the remaining CPU cores (lines 12 to 16). In other words, m and p values are proportionally adjusted with the relative acceleration factor. The process is repeated until a certain number of trials.

5 Performance Evaluation

5.1 Fast Ant System

To validate the approaches presented in this paper, the fast ant system (FANT) metaheuristic [14] has been considered. Basically, the major idea of FANT is to construct each solution (active ant) in a probabilistic way from the values of the decision variables in past searches by using a memory structure. To accelerate the convergence process, a local search algorithm is performed each time a solution is built. The process is repeated until a certain number of iterations is reached. The reinforcement parameter R has an impact during the intensification phase of the FANT metaheuristic.

The embedded local search is based on the selection of the best neighbor at each iteration. Such a selection mechanism accepting non-improving neighbors, will lead to cycles during the search process. Thereby, the number of local iterations has been restricted to $\frac{n}{2}$ (n is the instance size).

5.2 Application to the Quadratic Assignment Problem

The well-known QAP arises in many applications such as facility location or data analysis. The evaluation function has a $O(n^2)$ time complexity. In the next implementations, a neighborhood based on a pair-wise exchange ($\frac{n\times(n-1)}{2}$ neighbors) has been considered. For each iteration of a local search, $\frac{(n-2)\times(n-3)}{2}$ neighbors can be evaluated in $O(1)$ and $2n-3$ can be evaluated in $O(n)$.

From an implementation point of view, since calculations may be irregular according to the given neighbor, threads are reorganized in such a way that threads belonging to a same group of 32 threads (a.k.a. a warp) execute the same computation. In other words, groups of threads which perform $O(1)$ and $O(n)$ calculations are clearly separated. Such a mechanism allows reducing threads divergence due to conditional branches. Furthermore, to minimize the idle time due to irregular computations, $2n$ threads are associated with $O(n)$ calculations and $\frac{(n-1)}{2}$ threads execute $n \times O(1)$ calculations per local search. In this way, each thread block corresponds to one neighborhood evaluation.

5.3 Configuration

Experiments have been carried out on top of two different configurations. The first one is an Intel Core i7 930 with 4 cores cadenced at 2.8 Ghz using a NVIDIA GTX 480 graphic card (480 GPU cores). The second configuration is a bi-processor Intel Xeon E5520 with 2×4 cores cadenced at 2.26 Ghz using a Tesla C1060 (240 GPU cores). Since the first card provides on-chip memory for L1 cache memory, techniques to cache input data using the texture memory have only been applied to the second configuration. Posix threads have been considered for multi-core versions.

The average time has been measured in seconds for 30 runs, and acceleration factors are reported in comparison with a single CPU core. The standard deviation is not represented since its value is close to zero.

Table 1: Measures in terms of efficiency for the QAP using a pair-wise-exchange neighborhood. 4 FANT implementations on different architectures are considered.

Instance	**Core i7 930 2.8Ghz GeForce GTX 480 4 CPU cores 480 GPU cores**			**Xeon E5520 2.26Ghz Tesla C1060 8 CPU cores 240 GPU cores**		
	Multi-core	GPU	Heterogeneous	Multi-core	GPU	Heterogeneous
tai50a	$10.5_{\times\mathbf{2.4}}$	$2.1_{\times\mathbf{11.8}}$	$2.1_{\times\mathbf{11.9}\mid\times\mathit{14.2}}$	$7.5_{\times\mathbf{3.0}}$	$3.2_{\times\mathbf{7.0}}$	$2.3_{\times\mathbf{9.8}\mid\times\mathit{10.0}}$
tai60a	$17.8_{\times\mathbf{2.4}}$	$3.4_{\times\mathbf{12.7}}$	$3.3_{\times\mathbf{13.1}\mid\times\mathit{15.1}}$	$12.7_{\times\mathbf{3.1}}$	$5.4_{\times\mathbf{7.3}}$	$3.8_{\times\mathbf{10.4}\mid\times\mathit{10.4}}$
tai80a	$41.2_{\times\mathbf{2.6}}$	$7.5_{\times\mathbf{14.4}}$	$7.0_{\times\mathbf{15.4}\mid\times\mathit{17.0}}$	$29.4_{\times\mathbf{3.3}}$	$12.0_{\times\mathbf{8.1}}$	$8.5_{\times\mathbf{11.4}\mid\times\mathit{11.4}}$
tai100a	$81.7_{\times\mathbf{2.7}}$	$15.6_{\times\mathbf{14.0}}$	$13.9_{\times\mathbf{15.8}\mid\times\mathit{16.7}}$	$52.3_{\times\mathbf{3.8}}$	$22.3_{\times\mathbf{9.0}}$	$16.2_{\times\mathbf{12.3}\mid\times\mathit{12.8}}$
tai150b	$288.1_{\times\mathbf{2.7}}$	$73.7_{\times\mathbf{10.6}}$	$58.9_{\times\mathbf{13.2}\mid\times\mathit{13.3}}$	$138.7_{\times\mathbf{5.5}}$	$74.3_{\times\mathbf{10.3}}$	$50.3_{\times\mathbf{15.2}\mid\times\mathit{15.8}}$
tai256c	$1620.5_{\times\mathbf{2.7}}$	$382.7_{\times\mathbf{11.3}}$	$373.2_{\times\mathbf{11.6}\mid\times\mathit{14.0}}$	$610.1_{\times\mathbf{6.9}}$	$615.9_{\times\mathbf{6.9}}$	$351.9_{\times\mathbf{12.1}\mid\times\mathit{13.8}}$

5.4 Experimentation

The set of experiments consists in measuring the performance of proposed parallelization schemes. For doing this, four FANT versions have been implemented for the QAP. A CPU implementation using one single core, a multi-core version, a GPU implementation and another one using all the available heterogeneous resources. For all versions, 50 neighborhood evaluations (i.e. 50 active ants per global iteration) at a time have been considered. Regarding the semantic of the algorithms, there is no difference of the quality of solutions provided by both versions. The multi-core version does not intentionally utilize one CPU core in order to highlight the performance improvements of the heterogeneous version (since one core is associated with the GPU). The number of global iterations has been fixed to 10000, which corresponds to a realistic scenario in accordance with the algorithm convergence. Experimental results are reported in Table 1. The CPU column is not represented since the associated values can be deduced from the other columns.

Regarding the multi-core version (number of CPU cores minus one), the obtained acceleration factors grow with the instance size. For the first configuration using three cores, these speed-ups linearly vary from $\times 2.4$ to $\times 2.7$. This is not exactly the same phenomenon for the second configuration where acceleration factors alternate from $\times 3.0$ to $\times 6.9$. Indeed, for smaller instances, the overhead creation is significant in regards with the computational time. This is mainly due to the important number of threads to be created and synchronized (seven threads). But, as long as the size increases, the acceleration factor converges to the expected value.

For the GPU implementation, the obtained speed-ups are quite significant but relatively modest. They alternate from $\times 10.6$ to $\times 14.4$ for the first configuration, and from $\times 7$ to $\times 9.3$ for the second configuration. Such performance results are limited since most of move evaluations can be performed in $O(1)$. Therefore, the amount of computations is not enough to fully cover the memory access latency. Furthermore, the application is memory bound since non-coalescing accesses to the global memory drastically reduces the performance of the GPU implementation. This is due to high-misaligned accesses present in flows and distances matrices in QAP.

Regarding the heterogeneous version taking advantage of all CPU cores, the performance improvements in comparison with the GPU implementation are significant. Indeed, for the first configuration corresponding to three additional CPU cores, acceleration factors vary from $\times 11.6$ to $\times 15.8$, which corresponds to an improvement between 1% and 25%. For the second one with seven additional cores, better performance improvements between 39% and 75% (speed-ups varying from $\times 9.8$ to $\times 15.2$) can be observed.

To assess the efficiency of the heterogeneous version, potential acceleration factors are represented in italic in sub indices. These theoretical values are obtained by adding the speed-ups obtained for both multi-core and GPU versions. The performance difference, which occurs between the obtained results and the potential speed-up, is due to synchronization points between the GPU and the other CPU cores. One can clearly see that the acceleration factors obtained for the heterogeneous version are not so far from the expected theoretical ones. This is particularly the case for the second configuration containing more CPU cores. As a consequence, the heuristic for finding a parameters auto-tuning provides an efficient way to deal with load balancing for heterogeneous resources.

6 Conclusion

Hybrid metaheuristics having complementary behaviors allow improving the effectiveness and robustness in optimization. Their exploitation for solving real-world problems is possible only by using a great computational power. High-performance computing based on heterogeneous resources is recently revealed as an efficient way to use the huge amount of resources at disposal. However, the exploitation of parallel models is not trivial and many issues related to the task repartition between the GPU and multi-core architectures have to be faced.

In this paper, we have investigated on different parallelization strategies for hybrid metaheuristics on such heterogeneous resources. In the proposed parallelization approaches, the CPU manages the metaheuristic process and let the GPU be used as a coprocessor dedicated to intensive calculations. Thereafter, parts of these computations are distributed among the available CPU cores. Such a task repartition is provided by an efficient heuristic for parameters tuning.

The designed and implemented approaches have been experimentally validated on the QAP using the FANT metaheuristic. The evaluation of a neighboring solution in the QAP can be performed most of the time in constant time. As a result, for problems with modest GPU accelerations, the performance improvement provided by multi-core CPUs is particularly significant (up to 75% for eight CPU cores). In particular, we showed that our methodology enables gaining to a $\times 15$ factor in terms of acceleration compared with a single core architecture. A perspective of this work will be to implement the proposed approaches for other combinatorial optimization problems, in which the computational complexity of move evaluations is more prominent.

With the arrival of GPU resources in clusters of workstations and grids, the next objective is to examine the conjunction of GPU computing and distributed computing to fully and efficiently exploit the hierarchy of parallel models of metaheuristics. Indeed, since all processors are currently multi-core, performance of GPU-based algorithms might be drastically improved. The challenge will be to find the best mapping in terms of efficiency of the hierarchy of parallel models on the hierarchy of CPU-GPU resources provided by multi-level architectures.

Acknowledgments

Part of this research work was supported by HES-SO RCSO-TIC research grant 25018 CPUGPU.

References

1. Talbi, E.G.: A taxonomy of hybrid metaheuristics. J. Heuristics **8**(5) (2002) 541–564
2. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., mei W. Hwu, W.: Program optimization carving for gpu computing. J. Parallel Distributed Computing **68**(10) (2008) 1389–1401
3. Munawar, A., Wahib, M., Munetomo, M., Akama, K.: Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. Genetic Programming and Evolvable Machines **10** (2009) 391–415
4. Tsutsui, S., Fujimoto, N.: Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment. In Krasnogor, N., Lanzi, P.L., eds.: GECCO, ACM (2011) 1547–1554
5. Luong, T.V., Melab, N., Talbi, E.G.: Parallel hybrid evolutionary algorithms on gpu. In: IEEE Congress on Evolutionary Computation. (2010) 1–8
6. Talbi, E.G.: Metaheuristics: From design to implementation. Wiley (2009)
7. Wong, M.L., Wong, T.T., Fok, K.L.: Parallel evolutionary algorithms on graphics processing unit. In: Congress on Evolutionary Computation, IEEE (2005) 2286–2293
8. Mussi, L., Cagnoni, S., Daolio, F.: Gpu-based road sign detection using particle swarm optimization. In: ISDA, IEEE Computer Society (2009) 152–157
9. Bai, H., OuYang, D., Li, X., He, L., Yu, H.: Max-min ant system on gpu with cuda. In: Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control. ICICIC '09, Washington, DC, USA, IEEE Computer Society (2009) 801–804
10. Janiak, A., Janiak, W.A., Lichtenstein, M.: Tabu search on gpu. J. UCS **14**(14) (2008) 2416–2426
11. Zhu, W., Curry, J., Marquez, A.: Simd tabu search with graphics hardware acceleration on the quadratic assignment problem. International Journal of Production Research (2008)
12. Czapinski, M., Barnes, S.: Tabu search with two approaches to parallel flowshop evaluation on cuda platform. J. Parallel Distrib. Comput. **71**(6) (2011) 802–811
13. Luong, T.V., Melab, N., Talbi, E.G.: Gpu computing for parallel local search metaheuristic algorithms. IEEE Transactions on Computers **99**(PrePrints) (2011)
14. Taillard, E.D.: Fant: Fast ant system. Technical report (1998)