Research Report: GPU-based Approaches for Hybrid Metaheuristics

Thé Van Luong, Eric Taillard

1 Introduction

In combinatorial optimization, near-optimal algorithms such as metaheuristics allow to iteratively solve in a reasonable time NP-hard complex problems. Two main categories of metaheuristics are distinguished: population-based metaheuristics (P-metaheuristics) and solution-based metaheuristics (S-metaheuristics). P-metaheuristics are population-oriented as they manage a whole population of solutions, what confers them a good exploration power. Indeed, they allow to explore a large number of promising regions in the search space. On the contrary, S-metaheuristics such as local search algorithms work with a single solution which is iteratively improved by exploring its neighborhood in the solution space. Therefore, they are characterized by better local intensification capabilities.

Theoretical and experimental studies have shown that the hybridization between these two classes of metaheuristics improves the quality of provided solutions and the robustness of the metaheuristics [1]. Nevertheless, as it is generally CPU time-consuming it is not often fully exploited in practice. Indeed, experiments with hybrid metaheuristics are often stopped without convergence being reached. That is the reason why, in designing hybrid metaheuristics, there is often a compromise between the number of solutions to use and the computational complexity to explore it. To deal with such issues, only the use of parallelism allows to design efficient hybrid metaheuristics.

Recently, graphics processing units (GPU) have emerged as a new popular support for massively parallel computing [2], [3]. Such resources supply a great computing power, are energy-efficient, and unlike grids, they are highly available everywhere: laptops, desktops, clusters, etc. During many years, the use of GPU computing was dedicated to graphics and video applications. Its utilization has recently been extended to other application domains [4], [5] (e.g. scientific computing) thanks to the publication of the CUDA (Compute Unified Device Architecture) development toolkit that allows GPU programming in a C-like language [6].

With the emergence of standard programming languages on GPU and the arrival of compilers for these languages, combinatorial optimization on GPU has generated a growing interest. Historically, due to their embarrassingly parallel nature, P-metaheuristics such as evolutionary algorithms have been the first subject of parallelization on GPU architectures. Hence, previous approaches and implementations have been proposed for genetic algorithms [7], [8], particle swarm optimization [9], [10], ant colonies [11], [12], genetic programming [13], [14] and other evolutionary computation techniques [15], [16].

In comparison with previous works on population-based metaheuristics, the spread of solution-based metaheuristics on GPU does not occur at the same pace. Indeed, the parallelization on

T.V. Luong and E. Taillard are from both HEIG-VD, University of applied sciences of Western Switzerland, Yverdon-les-Bains, Switzerland (e-mail: the-van.luong@heig-vd.ch; eric.taillard@heig-vd.ch)

GPU architectures is harder, due to the improvement of a single solution (and not a population of solutions). In this purpose, we came up with the pioneering work in IEEE Transactions on Computers [17] for the re-design of the parallel evaluation of the neighborhood on GPU. We introduced the generation of neighbors on the GPU side to minimize the data transfers. Furthermore, we proposed to manage the commonly used structures in combinatorial optimization with the different available memories.

The main objective of this report is to deal with the re-design of hybrid metaheuristics on GPU architectures. Indeed, parallel hybrid metaheuristics for solving combinatorial optimization problems are significant challenges for GPU computing. To the best of our knowledge, only few research works have been investigated on hybrid metaheuristics on GPU [18], [19]. In [20], we proposed a parallelization scheme for hybrid evolutionary algorithms on GPU based on the parallel evaluation of the neighborhood. In the current research report, we extend this previous work with other parallelization approaches.

To validate the approaches presented in this paper, a metaheuristic based on fast ant systems (FANT) introduced by Taillard [21] has been considered and implemented on GPU. Basically, FANT incorporates a number of search strategies such as intensification (hybridization with a local search), diversification and learning mechanisms. As an example of application, the quadratic assignment problem (QAP) [22] has been considered.

The remainder of the paper is organized as follows: Section 2 highlights the principles of parallel models for metaheuristics. In Section 3, parallelization concepts for designing parallel hybrid metaheuristics on GPU are described. Section 4 reports the performance results obtained for the FANT metaheuristic applied to the quadratic assignment problem. Finally, a discussion and some conclusions of this work are drawn in Section 5.

2 Parallel Metaheuristics

2.1 Parallel Models of Metaheuristics

In general, evaluating a fitness function for each solution is frequently the most costly operation of the metaheuristic. That is the reason why, for hybrid metaheuristics, executing the iterative process of a S-metaheuristic (e.g. local search) on large neighborhoods requires a large amount of computational resources.

Consequently, parallelism arises naturally when dealing with a neighborhood, since each of the solutions belonging to it is an independent unit. Due to this, the performance of hybrid metaheuristics is significantly improved when running in parallel.

In this purpose, three major parallel models for metaheuristics can be distinguished [23]: solution-level, iteration-level and algorithmic-level (see Fig. 1).

- *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. In that case, the function can be viewed as an aggregation of a given number of partial functions.
- *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. At the beginning of each iteration, the master duplicates the solutions to be evaluated between parallel nodes. Each of them manages some candidates, and the results are returned back to the master. 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, indepen-

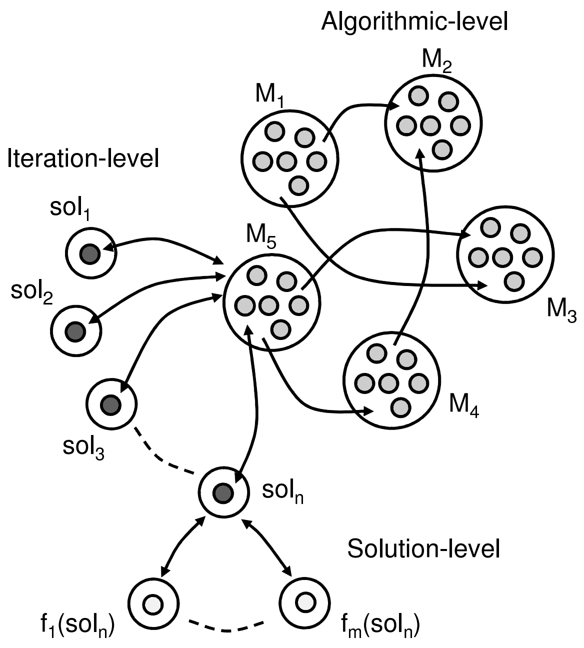

Fig. 1: Parallel models of metaheuristics.

dent or cooperative, start from the same or different solution(s), configured with the same or different parameters.

From a parallelization point of view, the solution-level model is problem-dependent and does not present many generic concepts. As a consequence, in this document, we will not deal with this parallel model.

2.2 Metaheuristics on Parallel and Distributed Architectures

During these two last decades, many parallel approaches and implementations have been proposed for metaheuristics. Some of them using massively parallel processors [24], clusters of workstations [25], [26] and shared memory or SMP machines [27], [28]. These contributions have been later revisited for large-scale computational grids [29].

These architectures often exploit the coarse-grained asynchronous parallelism based on work-stealing. This is particularly the case for computational grids. To overcome the problem of network latency, the grain size is often increased, limiting the degree of parallelism.

Recently, GPU accelerators have emerged as a powerful support for massively parallel computing. Indeed, these architectures offer a substantial computational horsepower and a remarkably high memory bandwidth compared to CPU-based architectures. For instance, the parallel evaluation of the solutions (iteration-level) is a Master-Worker and a problem-independent, regular data-parallel application. Therefore, GPU computing may be highly efficient in executing such synchronized parallel algorithms that involve regular computations and data transfers.

In general, for distributed architectures, the global performance in metaheuristics is limited by high communication latencies whilst it is just bounded by memory access latencies in GPU architectures. Indeed, when evaluating solutions in parallel, the main obstacle in distributed architectures is the communication efficiency. GPUs are not that versatile.

However, since the execution model of GPUs is purely SIMD, it may not be well-adapted for few irregular problems in which the execution time cannot be predicted at compile time and varies during the search. For instance, this happens when the evaluation cost of the objective function depends on the solution. When dealing with such problems in which the computations or the data transfers become irregular or asynchronous, parallel and distributed architectures such as COWs or computational grids may be more appropriate.

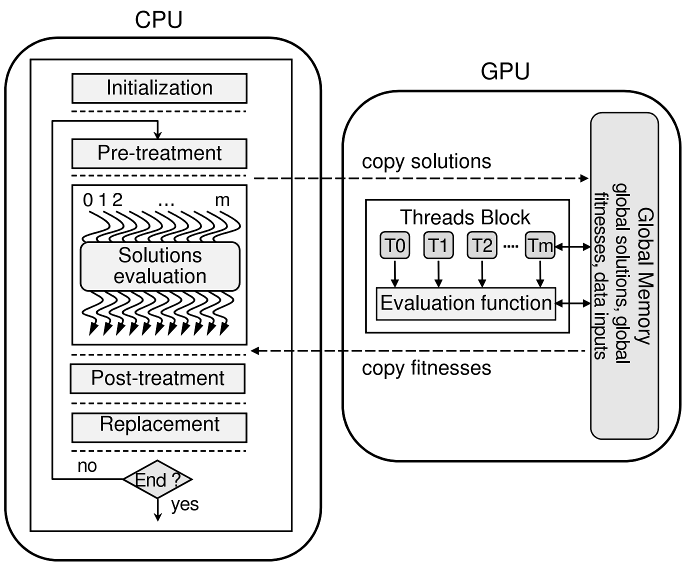

Fig. 2: Parallel evaluation of solutions on GPU (iteration-level). The evaluation of solutions is performed on GPU and the CPU executes the sequential part of the search process.

3 Design of Parallel Hybrid Metaheuristics on GPU

3.1 Parallel Evaluation of Solutions on GPU

As quoted above, the evaluation of solution candidates is often the most time-consuming part of metaheuristics. Thereby, it must be done in parallel in regards with the iteration-level parallel model. Hence, according to the Master-Worker paradigm, the idea is to evaluate the solutions in parallel on GPU.

To achieve this, the parallel iteration-level model has to be designed according to the data-parallel single program multiple data model of GPUs. As illustrated in Fig. 2, the CPU-GPU task partitioning is such that the CPU hosts and executes the whole serial 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. The granularity of each partition is determined by the number of threads per block.

This parallelization strategy has been widely used for P-metaheuristics on GPU especially for evolutionary algorithms due to their embarrassingly parallel workload (e.g in [7], [30], [31]).

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. Thereby, data transfer between CPU and GPU through the PCIe bus might be a serious bottleneck in the performance of GPU applications. For metaheuristics, these copies are essentially 1) the solutions to be evaluated and 2) their resulting fitnesses.

3.2 The Proposed Algorithm for S-metaheuristics

When it comes to parallelization, the optimization of data transfers is more prominent for S-metaheuristics such as local search algorithms. Furthermore, as previously said, the execution of the S-metaheuristic of the hybrid algorithm constitutes the time-consuming operation. Hence, when designing hybrid metaheuristics, the focus is on the embedded S-metaheuristic. In this purpose, we have contributed in [17] for the parallel evaluation of neighborhoods for local search algorithms. Algorithm 1 sums up the different steps required for the parallelization of S-metaheuristics on GPU.

Algorithm 1 S-metaheuristic Template on GPU

```
1:  Choose an initial solution
2:  Evaluate the solution
3:  Specific initializations
4:  Allocate problem inputs on GPU memory
5:  Allocate a solution on GPU memory
6:  Allocate a fitnesses structure on GPU memory
7:  Allocate additional structures on GPU memory
8:  Copy problem inputs on GPU memory
9:  Copy the initial solution on GPU memory
10: Copy additional structures on GPU memory
11: repeat
12:     for each neighbor in parallel do
13:         Evaluation of the candidate solution
14:         Insert the resulting fitness into the fitnesses structure
15:     end for
16:     Copy the fitnesses structure on CPU memory
17:     Specific selection strategy on the fitnesses structure
18:     Specific post-treatment
19:     Copy the chosen solution on GPU memory
20:     Copy additional structures on GPU memory
21: until a stopping criterion satisfied
```

First of all, at initialization stage, memory allocations on GPU are made: data inputs and candidate solution of the given problem (lines 4 and 5). A structure has to be allocated for storing the results of the evaluation of each neighbor (fitnesses structure) (line 6). Additional structures, which are problem-dependent, might be allocated to facilitate the computation of neighbor evaluations (line 7). Second, problem data inputs, initial candidate solution and additional structures associated with this solution have to be copied onto the GPU (lines 8 to 10). Third, comes the parallel iteration-level on GPU, in which each neighboring solution is generated (parallelism control), evaluated (memory management) and copied into the fitnesses structure (from lines 12 to 15). Fourth, the order in which candidate neighbors are evaluated is undefined, and then the fitnesses structure has to be copied to the host CPU (line 16). Then a solution selection strategy is applied to this structure (line 17): the exploration of the neighborhood fitnesses structure is carried out by the CPU in a sequential way. Finally, after a new candidate has been selected, this latter and its additional structures are copied to the GPU (lines 19 and 20). The process is repeated until a stopping criterion is satisfied.

This methodology is well-adapted to any deterministic local search methods. Its applicability does not stand on any assumption.

3.3 Additional Data Transfer Optimization

In some S-metaheuristics such as hill climbing or variable neighborhood descent, the selection operates on the minimal/maximal fitness for finding the best solution. Therefore, only one value of the fitnesses structure may be merely copied from the GPU to the CPU (line 16 of Algorithm 1). Such mechanism may lead to an increase of the obtained performance. However, since read/write operations on memory are performed in an asynchronous manner, finding the appropriate minimal/maximal fitnesses on GPU is not straightforward. Indeed, traditional parallel techniques such as semaphores which imply the global synchronization (via atomic

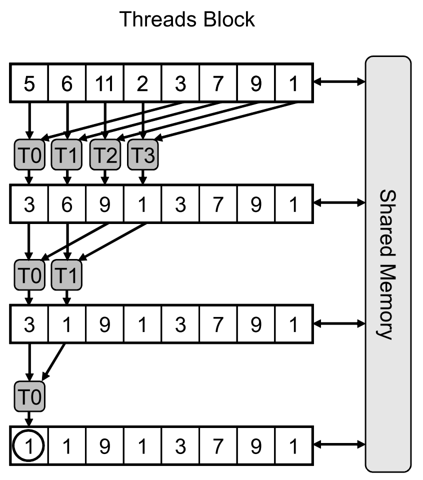

Fig. 3: Reduction operation to find the minimum of each block. Local synchronizations are performed between threads of a same block via the shared memory.

operations) of thousands of threads can drastically lead to diminished performance. To deal with this issue, adaptation of parallel reduction techniques for each thread block must be considered (see Fig. 3).

Algorithm 2 gives a template of the parallel reduction for a thread block (partition of the neighborhood). Basically, each thread loads one element from global to shared memory (lines 1 and 2). At each loop iteration, elements of the array are compared by pairs (lines 3 to 7). Then, by using local synchronizations between threads in a given block via the shared memory, one can find the minimum/maximum of a given array since threads operate at different memory addresses. For the sake of simplicity, the template is given for dealing with a neighborhood size which is a power of two, but adaptation of the template for the general case is straightforward. The complexity of such an algorithm is in $O(log_2(n))$ where n is the size of each thread block. If several iterations are performed on reduction kernels, the minimum of all the neighbors can be found. Thereby, the GPU reduction kernel makes it possible to get the minimum/maximum of each block of threads.

Algorithm 2 Reduction kernel on the fitnesses structure.

```
Require: input_fitnesses;
 1: shared[thread_id] := input_fitnesses[id];
 2: local synchronization;
 3: for i := nbThreadsPerBlock/2 ; i > 0; i := i / 2 do
 4:    if thread_id < i then
 5:       shared[thread_id] := compare(shared[thread_id], shared[thread_id + i]);
 6:       local synchronization;
 7:    end if
 8: end for
 9: if thread_id = 0 then
10:    output_fitnesses[blockId] := shared[0];
11: end if
Ensure: output_fitnesses;
```

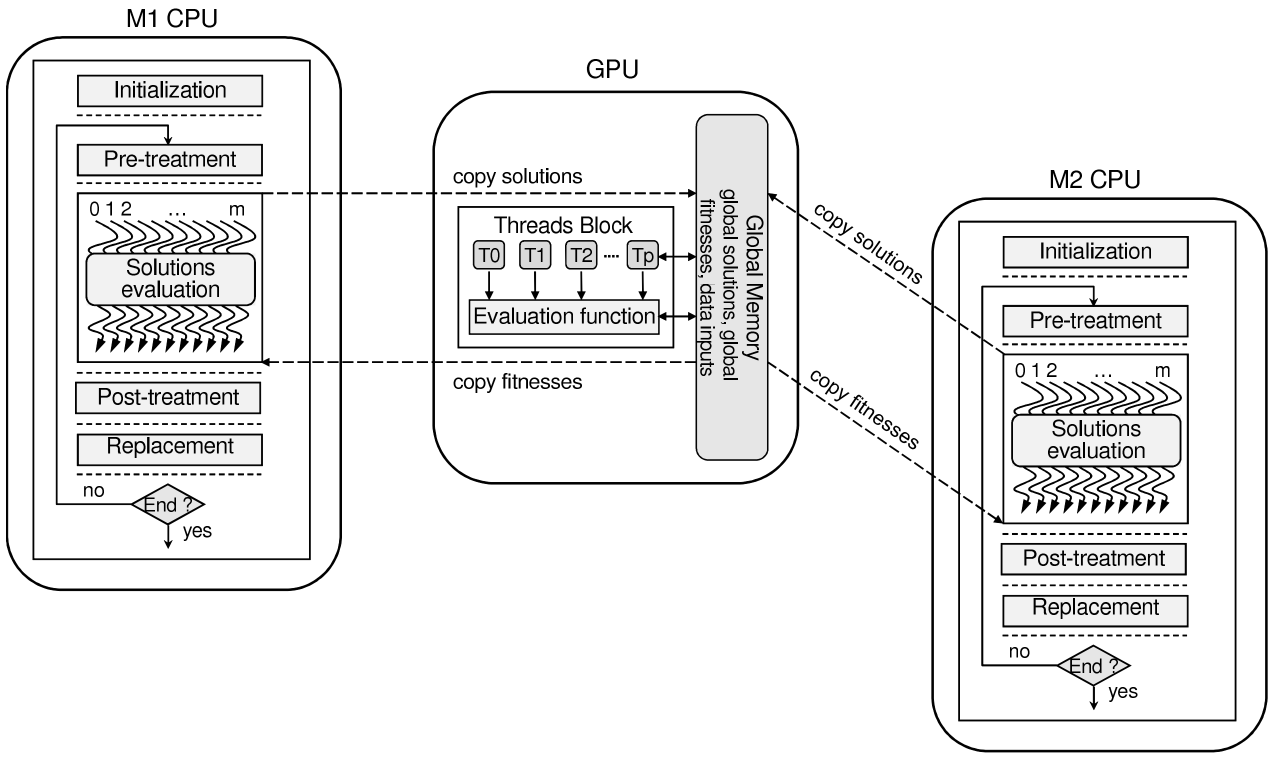

Fig. 4: Multiple evaluations of neighborhoods on GPU. The evaluation of solutions of all S-metaheuristics is performed on GPU and the CPU executes the sequential part of the search process.

3.4 Parallelization Schemes for Hybrid Metaheuristics

The parallelization approaches presented in Section 3.2 and 3.3 stand for one S-metaheuristic on GPU. For designing hybrid metaheuristics that involve a population of solutions, multiple executions of S-metaheuristics on GPU have to be considered. For doing this, there are fundamentally two parallelization schemes for hybrid metaheuristics:

- *One neighborhood evaluation on GPU*. This approach consists in evaluating one neighborhood (a set of solutions) at a time on GPU. It can be seen as an acceleration model which does not change the semantic of the original algorithm. According to Fig. 2, a possible interpretation could be to reiterate the whole process (i.e. 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 (solutions) executed on GPU might not be enough to cover the memory access latency for few optimization problems. Furthermore, since each kernel execution on GPU (a neighborhood evaluation) has a creation overhead, such an approach could have a negative impact on the performance of the hybrid metaheuristic at hand.
- *Many neighborhood evaluations on GPU*. In the second approach, many neighborhoods are evaluated at a time on GPU. Fig. 4 illustrates this concept for two S-metaheuristics. In this example, the two neighborhoods are evaluated in a same time on GPU, and the results are returned back to the CPU in a synchronous way. This could be generalized to many neighborhoods. Such a parallelization strategy deals with the issues encountered in the first approach i.e. there are enough calculations to keep the GPU multiprocessors busy. However, in this second approach, homogeneous embedded S-metaheuristics are required. As a result, such a mechanism may clearly modify the semantic of the original algorithm.

4 Performance Evaluation

4.1 Fast Ant System

As an illustration of a hybrid metaheuristic, the FANT metaheuristic has been considered (see Algorithm 3). Basically, the major idea of FANT is to construct each solution (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 (i.e. a certain number of ants has been processed). The reinforcement parameter R has an impact during the intensification phase of the FANT metaheuristic. Details of the algorithm can be found in [21].

Algorithm 3 FANT pseudo-code

```
1: InitMemory($m_0$);
2: $t$ := 0;
3: repeat
4:     $s(t)$ := GenerateSolution($m(t)$);
5:     $s(t)$ := ApplyLocalSearch($s(t)$);
6:     $m(t+1)$ := UpdateMemory($s(t), R, m(t)$)
7:     $t$ := $t+1$;
8: until a certain number of iterations.
```

A profiling of a FANT implementation on CPU highlights that more than 95% of the execution time is dedicated to the local search algorithm. Hence, the parallelization of FANT is within the scope of the parallelization schemes on GPU presented in Section 3.

4.2 Application to the Quadratic Assignment Problem

The quadratic assignment problem [32] arises in many applications such as facility location or data analysis. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ matrices of positive integers. Finding a solution of the quadratic assignment problem is equivalent to finding a permutation $\pi = (1, 2, \dots, n)$ that minimizes the objective function:

$$z(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\pi(i)\pi(j)}$$

The problem has been implemented using a permutation representation. The considered instances are the Taillard instances proposed in [33], [34]. The evaluation function has a $O(n^2)$ time complexity where n is the instance size. In the next implementations, a neighborhood based on a pair-wise exchange ($\frac{n\times(n-1)}{2}$ neighbors) has been considered. Hence, for each iteration of a local search, $\frac{(n-2)\times(n-3)}{2}$ neighbors can be evaluated in $O(1)$ (Δ evaluations) and $2 \times (n-3)$ can be evaluated in $O(n)$. The requirement is a structure which stores previous Δ evaluations in a quadratic space complexity.

From an implementation point of view, a natural approach would be to consider a GPU kernel execution of $\frac{n\times(n-1)}{2}$ threads (i.e. one thread per neighbor). However, for this problem such a mechanism is not appropriate since calculations may be irregular according to the given neighbor. To deal with this drawback, threads are reorganized in such a way that threads belonging to a same group of 32 threads (a.k.a. a warp) perform the same computation. In other words, groups of threads which perform $O(1)$ and $O(n)$ calculations are clearly separated. Such a mechanism allows to reduce threads divergence due to conditional branches. Furthermore, to minimize the idle time due to irregular computations, we have proposed to take $2n$ threads to perform $O(n)$ calculations and $\frac{(n-1)}{2}$ threads to execute $n \times O(1)$ calculations per local search.

TABLE 1: Comparison of parallelization schemes for hybridization in terms of efficiency for the QAP using a pair-wise-exchange neighborhood.

$^{\pm}$ represents the standard deviation regarding the execution time, and $_{\times}$ is the acceleration factor in comparison with a CPU version using one single core.

Instance	**FANT + hill climbing** CPU sequential	**FANT + hill climbing** One neighborhood evaluation on GPU	**FANT + hill climbing** Many neighborhood evaluations on GPU
tai50a	$161.5^{\pm 1.4}$	$193.3^{\pm 1.8}_{\times \mathbf{0.8}}$	$18.7^{\pm 0.3}_{\times \mathbf{8.6}}$
tai60a	$272.1^{\pm 2.6}$	$204.5^{\pm 2.4}_{\times \mathbf{1.3}}$	$29.4^{\pm 0.7}_{\times \mathbf{9.2}}$
tai80a	$658.7^{\pm 7.3}$	$348.9^{\pm 1.6}_{\times \mathbf{1.9}}$	$58.2^{\pm 1.1}_{\times \mathbf{11.3}}$
tai100a	$1294.4^{\pm 63.2}$	$530.2^{\pm 5.2}_{\times \mathbf{2.4}}$	$117.2^{\pm 6.3}_{\times \mathbf{11.0}}$
tai100b	$1802.3^{\pm 81.4}$	$815.8^{\pm 8.8}_{\times \mathbf{2.2}}$	$174.3^{\pm 6.2}_{\times \mathbf{10.3}}$
tai150b	$7088.3^{\pm 612.2}$	$1987.5^{\pm 48.1}_{\times \mathbf{3.6}}$	$859.4^{\pm 54.7}_{\times \mathbf{8.2}}$
tai64c	$111.7^{\pm 2.7}$	$40.0^{\pm 1.1}_{\times \mathbf{2.8}}$	$7.8^{\pm 0.6}_{\times \mathbf{14.3}}$
tai256c	$9185.7^{\pm 210.1}$	$803.9^{\pm 36.2}_{\times \mathbf{11.4}}$	$789.8^{\pm 39.3}_{\times \mathbf{11.6}}$

4.3 Experimentation

Experiments have been carried out on top of an Intel Core i7 930 2.8 Ghz using a NVIDIA GTX 480 graphic card (480 GPU cores cadenced at 700 Mhz). Since this card provides on-chip memory for L1 cache memory, techniques to cache input data using the texture memory have not been applied. For each produced implementation, the *nvcc* compiler of the CUDA toolkit version 4.0 and the Intel *icc* compiler version 12.1 have been used. The average time has been measured in seconds for 30 runs including the associated standard deviation, and acceleration factors are reported in comparison with a single CPU core.

4.3.1 Comparison of Parallelization Schemes for Hybridization

A first set of experiments consists in comparing the performance of the two parallelization schemes presented in Section 3.4. For doing this, three FANT implementations using a hill climbing algorithm have been implemented for the quadratic assignment problem. A CPU implementation using one single core, a GPU implementation using one neighborhood evaluation on GPU per local search iteration, and another one using 50 neighborhood evaluations (i.e. 50 ants are evolving in parallel) at a time have been considered. The parallelization scheme used for the first GPU implementation does not change the semantic of the CPU sequential version (i.e. both produced outputs are identical). The number of FANT iterations has been fixed to 75000, which corresponds to a realistic scenario in accordance with the algorithm convergence. Experimental results are reported in Table 1.

For the first GPU version ($\frac{n\times(n-1)}{2}$ neighbors), the obtained acceleration factors are rather low for most instances. They vary from $\times 0.8$ to $\times 3.6$ for taixxa and taixxb instances. Indeed, since the neighborhood size is relatively small and most calculations can be performed in $O(1)$, the number of threads per block is not enough to fully cover the memory access latency. Such a phenomenon less occurs for the tai256c since the neighborhood size is more important (acceleration factor of $\times 11.4$).

It also less appears in the second parallelization scheme, in which many neighborhood evaluations are considered. Indeed, in the second GPU implementation, multiple ants are evaluated at the same time ($50 \times \frac{n\times(n-1)}{2}$ neighbors per iteration). Thereby, the amount of computations per

TABLE 2: Measures in terms of efficiency for different local search algorithms using a pair-wise-exchange neighborhood. The quadratic assignment problem is considered.

$^{\pm}$ represents the standard deviation regarding the execution time, and $_{\times}$ is the acceleration factor in comparison with a CPU version using one single core.

Instance	**FANT + hill climbing**			**FANT + best neighbor**			**FANT + tabu search**	
	CPU	GPU	GPUR	CPU	GPU	GPUR	CPU	GPU
tai50a	$161.5^{\pm 1.4}$	$18.7^{\pm 0.3}_{\times \mathbf{8.6}}$	$23.4^{\pm 0.2}_{\times \mathbf{6.9}}$	159.5	$15.3_{\times \mathbf{10.4}}$	$23.2_{\times \mathbf{6.9}}$	140.1	$32.4_{\times \mathbf{4.3}}$
tai60a	$272.1^{\pm 2.6}$	$29.4^{\pm 0.7}_{\times \mathbf{9.2}}$	$31.5^{\pm 0.3}_{\times \mathbf{8.6}}$	275.9	$25.1_{\times \mathbf{11.0}}$	$31.7_{\times \mathbf{8.7}}$	199.7	$43.3_{\times \mathbf{4.6}}$
tai80a	$658.7^{\pm 7.3}$	$58.2^{\pm 1.1}_{\times \mathbf{11.3}}$	$49.8^{\pm 0.6}_{\times \mathbf{13.2}}$	683.7	$54.6_{\times \mathbf{12.5}}$	$49.7_{\times \mathbf{13.7}}$	362.4	$69.2_{\times \mathbf{5.2}}$
tai100a	$1294.4^{\pm 63.2}$	$117.2^{\pm 6.3}_{\times \mathbf{11.0}}$	$90.5^{\pm 3.1}_{\times \mathbf{14.3}}$	1401.9	$121.8_{\times \mathbf{11.5}}$	$105.6_{\times \mathbf{13.3}}$	587.8	$108.4_{\times \mathbf{5.4}}$
tai100b	$1802.3^{\pm 81.4}$	$174.3^{\pm 6.2}_{\times \mathbf{10.3}}$	$127.1^{\pm 4.6}_{\times \mathbf{14.1}}$	1397.5	$120.6_{\times \mathbf{11.6}}$	$105.2_{\times \mathbf{13.3}}$	577.6	$104.2_{\times \mathbf{5.5}}$
tai150b	$7088.3^{\pm 612.2}$	$859.4^{\pm 54.7}_{\times \mathbf{8.2}}$	$654.1^{\pm 51.3}_{\times \mathbf{10.8}}$	5127.2	$586.7_{\times \mathbf{8.7}}$	$490.5_{\times \mathbf{10.4}}$	1375.5	$286.2_{\times \mathbf{4.8}}$
tai64c	$111.7^{\pm 2.7}$	$7.8^{\pm 0.6}_{\times \mathbf{14.3}}$	$6.6^{\pm 0.4}_{\times \mathbf{16.9}}$	331.7	$27.8_{\times \mathbf{11.9}}$	$29.2_{\times \mathbf{11.3}}$	217.1	$38.3_{\times \mathbf{5.7}}$
tai256c	$9185.7^{\pm 210.1}$	$789.8^{\pm 39.3}_{\times \mathbf{11.6}}$	$733.7^{\pm 21.1}_{\times \mathbf{12.5}}$	29044.6	$2528.6_{\times \mathbf{11.5}}$	$2039.8_{\times \mathbf{14.2}}$	4537.5	$762.5_{\times \mathbf{5.9}}$

GPU kernel is more prominent in this implementation. As a result, in comparison with the first version, the obtained results are significantly better (speed-ups varying between $\times 8.2$ to $\times 14.3$).

As a conclusion, for the quadratic assignment problem in which most neighboring solutions can be evaluated in $O(1)$, the second parallelization scheme for the hybrid metaheuristic may be more appropriate than the first one in terms of performance. Nevertheless, as previously said, such a mechanism totally modifies the semantic of the original FANT algorithm.

4.3.2 Performance of Embedded Local Search Algorithms

A second set of experiments consists in assessing the performance of different local search algorithms used in FANT. In the following implementations, three embedded local search algorithms have been considered on CPU and GPU. The first one is the hill climbing previously seen. The second algorithm is a local search based on the selection of the best neighbor at each iteration, and the last one is a tabu search. For the two first methods, one variant using the reduction mechanism (GPUR) to find the minimal fitness (see Section 3.3) has also been considered. The number of FANT iterations has been fixed to 75000 for the two first versions and to 1000 for the tabu search. The number of local iterations for the second algorithm has been set to $n/2$ and to 2000 for the tabu search. The standard deviation is not represented for the two last algorithms since its value is near 0. The obtained results for the different implementations are reported in Table 2.

Regarding the performance for the second algorithm (FANT + best neighbor), the acceleration factors of the first version are similar to those obtained for the hill climbing (speed-ups alternate from $\times 8.7$ and $\times 12.5$). This is not the case for the tabu search since additional post-treatments (e.g. tabu list management on CPU) are performed (acceleration factors varying from $\times 4.3$ to $\times 5.9$).

Regarding the variant for the two first algorithms using a reduction mechanism (GPUR), from the instance tai80a, finding the minimal fitness on GPU provides a performance improvement between 15% and 30%. Indeed, such an approach reduces the data transfers from the GPU to the CPU to one fitness.

Another observation from these results comes from the instance taixxc. In a general manner, the best performance of the different algorithms are obtained for these instances (e.g. a speed-up of $\times 16.9$ for the instance tai64c for the first algorithm using GPUR). Such a performance difference can be explained by the nature of these instances, which are mainly two matrices constituted by 0 and 1 values.

5 Conclusion

Hybrid metaheuristics having complementary behaviors allow to improve the effectiveness and robustness in combinatorial optimization. Their exploitation for solving real-world problems is possible only by using a great computational power. High-performance computing based on the use of computational GPUs 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 GPU hierarchical management of this architecture have to be considered.

In this research report, we have investigated on different parallelization approaches for hybrid metaheuristics on GPU. In the proposed parallelization strategies, the CPU manages the metaheuristic process and let the GPU be used as a coprocessor dedicated to intensive calculations. The designed and implemented approaches have been experimentally validated on the quadratic assignment problem using the FANT metaheuristic. In particular, we showed that our methodology enables to gain up to a factor between $\times 10$ and $\times 15$ in terms of acceleration compared with a single core architecture. Such an improvement is quite significant but not really impressive. This is mainly due to the evaluation of a neighboring solution in the quadratic assignment problem, which can be often performed in constant time. A perspective of this work will be to implement the proposed approaches for other combinatorial optimization problems, in which the evaluation of solutions is more prominent.

Furthermore, GPU-accelerated algorithms designed in this report only exploit a single CPU core. With the arrival of GPU resources in COWs 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, all processors are nowadays multi-core and when coupled with GPU devices hybrid or heterogeneous computing, which is definitely a new trend of parallel computing, 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.

References

[1] E.-G. Talbi, "A taxonomy of hybrid metaheuristics," *J. Heuristics*, vol. 8, no. 5, pp. 541–564, 2002.

[2] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu, "Program optimization carving for gpu computing," *J. Parallel Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.

[3] J. D. Owens, M.Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[7] M. L. Wong, T.-T. Wong, and K.-L. Fok, "Parallel evolutionary algorithms on graphics processing unit," in *Congress on Evolutionary Computation*. IEEE, 2005, pp. 2286–2293.

[8] Q. Yu, C. Chen, and Z. Pan, "Parallel genetic algorithms on programmable graphics hardware," in *Lecture Notes in Computer Science 3612*. Springer, 2005, p. 1051.

[9] L. Mussi, S. Cagnoni, and F. Daolio, "Gpu-based road sign detection using particle swarm optimization," in *ISDA*. IEEE Computer Society, 2009, pp. 152–157.

[10] Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *IEEE Congress on Evolutionary Computation*. IEEE, 2009, pp. 1493–1500.

[11] H. Bai, D. OuYang, X. Li, L. He, and H. Yu, "Max-min ant system on gpu with cuda," in *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*, ser. ICICIC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 801–804. [Online]. Available: http://dx.doi.org/10.1109/ICICIC.2009.255

[12] N. A. Sinnott-Armstrong, C. S. Greene, and J. H. Moore, "Fast genome-wide epistasis analysis using ant colony optimization for multifactor dimensionality reduction analysis on graphics processing units," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 215–216. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830523

[13] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Proceedings of the 10th European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, vol. 4445. Springer, 2007, pp. 90–101.

[14] D. M. Chitty, "A data parallel approach to genetic programming using programmable graphics hardware," in *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, vol. 2. London: ACM Press, 2007, pp. 1566–1573. [Online]. Available: http://www.cs.bham.ac.uk/ wbl/biblio/gecco2007/docs/p1566.pdf

[15] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm." in *Parallel and Distributed Computing, Applications and Technologies*, ser. PDCAT. IEEE Computer Society, 2009, pp. 457–462.

[16] L. de P. Veronese and R. A. Krohling, "Differential evolution algorithm on the gpu with c-cuda," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–7.

[17] T. V. Luong, N. Melab, and E.-G. Talbi, "Gpu computing for parallel local search metaheuristic algorithms," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2011.

[18] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 391–415, 2009, 10.1007/s10710-009-9091-4. [Online]. Available: http://dx.doi.org/10.1007/s10710-009-9091-4

[19] S. Tsutsui and N. Fujimoto, "Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment," in *GECCO*, N. Krasnogor and P. L. Lanzi, Eds. ACM, 2011, pp. 1547–1554.

[20] T. V. Luong, N. Melab, and E.-G. Talbi, "Parallel hybrid evolutionary algorithms on gpu," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[21] E. D. Taillard, "Fant: Fast ant system," Tech. Rep., 1998.

[22] T. C. Koopmans and M. Beckmann, "Assignment Problems and the Location of Economic Activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957. [Online]. Available: http://www.jstor.org/stable/1907742

[23] E.-G. Talbi, *Metaheuristics: From design to implementation*. Wiley, 2009.

[24] J. Chakrapani and J. Skorin-Kapov, "Massively Parallel Tabu Search for the Quadratic Assignment Problem," *Annals of Operations Research*, vol. 41, pp. 327–341, 1993.

[25] T. Crainic, M. Toulouse, and M. Gendreau, "Parallel Asynchronous Tabu Search for Multicommodity Location-Allocation with Balancing Requirements," *Annals of Operations Research*, vol. 63, pp. 277–299, 1995.

[26] B.-L. Garcia, J.-Y. Potvin, and J.-M. Rousseau, "A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints," *Computers & OR*, vol. 21, no. 9, pp. 1025–1033, 1994.

[27] T. James, C. Rego, and F. Glover, "A cooperative parallel tabu search algorithm for the quadratic assignment problem," *European Journal of Operational Research*, vol. 195, pp. 810–826, 2009.

[28] A. Bevilacqua, "A methodological approach to parallel simulated annealing on an smp system," *J. Parallel Distrib. Comput.*, vol. 62, no. 10, pp. 1548–1570, 2002.

[29] A.-A. Tantar, N. Melab, and E.-G. Talbi, "A comparative study of parallel metaheuristics for protein structure prediction on the computational grid," in *IPDPS*. IEEE, 2007, pp. 1–10.

[30] W. Zhu, "A study of parallel evolution strategy: pattern search on a gpu computing platform," in *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, ser. GEC '09. New York, NY, USA: ACM, 2009, pp. 765–772. [Online]. Available: http://doi.acm.org/10.1145/1543834.1543939

[31] R. Arora, R. Tulshyan, and K. Deb, "Parallelization of binary and real-coded genetic algorithms on gpu using cuda," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.

[32] R. E. Burkard, E. Çela, G. Rote, and G. J. Woeginger, "The quadratic assignment problem with a monotone anti-monge and a symmetric toeplitz matrix: Easy and hard cases," *Math. Program.*, vol. 82, pp. 125–158, 1998.

[33] É. D. Taillard, "Robust taboo search for the quadratic assignment problem," *Parallel Computing*, vol. 17, no. 4-5, pp. 443–455, 1991.

[34] ric D. and Taillard, "Comparison of iterative searches for the quadratic assignment problem," *Location Science*, vol. 3, no. 2, pp. 87 – 105, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0966834995000086

[35] *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*. IEEE, 2010.