GPU-based Multi-start Local Search Algorithms

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

INRIA Dolphin Project / Opac LIFL CNRS
40 avenue Halley, 59650 Villeneuve d'Ascq Cedex FRANCE.
The-Van.Luong@inria.fr, [Nouredine.Melab, El-Ghazali.Talbi]@lifl.fr

Abstract. In practice, combinatorial optimization problems are complex and computationally time-intensive. Local search algorithms are powerful heuristics which allow to significantly reduce the computation time cost of the solution exploration space. In these algorithms, the multi-start model may improve the quality and the robustness of the obtained solutions. However, solving large size and time-intensive optimization problems with this model requires a large amount of computational resources. GPU computing is recently revealed as a powerful way to harness these resources. In this paper, the focus is on the multi-start model for local search algorithms on GPU. We address its re-design, implementation and associated issues related to the GPU execution context. The preliminary results demonstrate the effectiveness of the proposed approaches and their capabilities to exploit the GPU architecture.

Keywords: GPU-based metaheuristics, multi-start on GPU.

1 Introduction

Over the last years, interest in metaheuristics (generic heuristics) has risen considerably in the field of optimization. Indeed, plenty of hard problems in a wide range of areas including logistics, telecommunications, biology, etc., have been modeled and tackled successfully with metaheuristics. Local search (LS) algorithms are a class of metaheuristics which handle with a single solution iteratively improved by exploring its neighborhood in the solution space. Different parallel models have been proposed in the literature for the design and implementation of LSs [1]. The multi-start model consists in executing in parallel many LSs in an independent/cooperative manner. This mechanism may provide more effective, diversified and robust solutions.

Nevertheless, although LS methods have provided very powerful search algorithms, problems in practice are becoming more and more complex and CPU time-intensive and their resolution requires to harness more and more computational resources. In parallel, the recent advances in hardware architecture allow to provide such required tremendous computational power through GPU infrastructures. This new emerging technology is indeed believed to be extremely useful to speed up many complex algorithms. However, the exploitation of such computational infrastructures in metaheuristics is not straightforward.

Indeed, several scientific challenges mainly related to the hierarchical memory management or to the execution context have to be faced. The major issues are the efficient distribution of data processing between the CPU and the GPU, the thread synchronization, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc. The main objective of our research work is to deal with such issues for the re-design of parallel metaheuristics models to allow solving of large scale optimization problems on GPU architectures. In [2, 3], we have proposed to re-design the parallel evaluation of the neighborhood model for LSs on GPU. To go on this way, the main objective of this paper is to deal with the well-known multi-start model on GPU architectures where many LSs are executed in parallel.

We deal with the entire re-design of the multi-start model on GPU by taking into account the particular features related to both the LS process and the GPU computing. More exactly, we provide two different general schemes for building efficient multi-start LSs on GPU. The first scheme combines the multi-start model with the parallel evaluation of the neighborhood on GPU previously mentioned above. In the second scheme, the search process of each LS algorithm is fully distributed on GPU. The advantage of the full distribution of the search process on GPU is to reduce CPU/GPU memory copy latency. We will essentially focus on this approach throughout this paper.

Despite the fact that the second scheme for the multi-start model has already been applied in some previous works in the context of the tabu search on GPU [4,5], to the best of our knowledge, it has never been widely investigated in terms of 1) reproducibility for any other LS algorithm and 2) memory management. Indeed, the contribution of this paper is to provide a general methodology for the design of multi-start LSs on GPU applicable to any class of LS algorithms such as hill climbing, tabu search or simulated annealing. Furthermore, a particular focus is made on finding efficient associations between the different available memories and the data commonly used in the multi-start LS algorithms.

The remainder of the paper is organized as follows: on the hand, Section 2 highlights the principles of LS parallel models. On the other hand, a brief review of the GPU architecture is also depicted. Section 3 presents a methodology for the design and the implementation of parallel multi-start LS methods on GPU. The performance results obtained for the associated implementations are reported in Section 4. Finally, a discussion and some conclusions of this work are drawn in Section 5.

2 Parallel Local Search Algorithms and GPU Computing

2.1 Parallel Models of LS Algorithms

For non-trivial problems, executing the iterative process of a simple LS on large neighborhoods requires a large amount of computational resources. Consequently, a variety of algorithmic issues are being studied to design efficient LS heuristics. 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 LS algorithms is particularly improved when running in parallel. Parallel design and implementation of metaheuristics have been studied as well on different architectures [6–8].

Basically, three major parallel models for LS heuristics can be distinguished: solution-level, iteration-level and algorithmic-level.

- *Solution-level Parallel Model.* A focus is made on the parallel evaluation of a single solution. Problem-dependent operations performed on solutions are parallelized. 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. Exploration and evaluation of the neighborhood are made in parallel. At the beginning of each iteration, each parallel node manages some candidates and the results are returned back to the master. An efficient execution is often obtained particularly when the evaluation of each solution is costly.
- *Algorithmic-level Parallel Model.* Several LS algorithms 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.

The solution-level model is problem-dependent and does not present many generic concepts. In this paper, we will focus on the multi-start model which is an instantiation of the algorithmic-level model where LS algorithms are all homogeneous.

2.2 GPU Computing

GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computation. A complete review of GPU architecture can be found in [9].

In general-purpose computing on graphics processing units, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Memory transfer from the CPU to the GPU device memory is a (a)synchronous operation which is time consuming. Bus bandwidth and latency between the CPU and the GPU can significantly decrease the performance of the search, so data transfers must be minimized.

Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple processors simultaneously execute the same program on different data. For achieving this, the concept of kernel is defined. The kernel is a function callable from the host and executed on the specified device by several processors in parallel.

This kernel handling is dependent of the general-purpose language. For instance, CUDA [10] or OpenCL [11] are parallel computing environments which provide an application programming interface. These toolkits introduce a model of threads which provides an easy abstraction for single-instruction and multiple-data (SIMD) architecture.

Regarding their spatial organization, threads are organized within so called thread blocks. A kernel is executed by multiple equally threaded blocks. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor, while different thread blocks can be assigned to different multiprocessors.

From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors are used according to the SPMD model, threads share the same code and have access to different memory areas. Basically, the communication between the CPU host and its device is done through the global memory.

3 Design and implementation of multi-start local search algorithms on GPU

With the recent advances in parallel computing particularly based on GPU computing, the multi-start model has to be re-visited from the design and implementation points of view. In this section, we propose multiple deployment schemes of the multi-start model for LS algorithms on GPU.

3.1 Multi-start Local Search Algorithms Based on the Iteration-level

In [2, 3], we have proposed the design and the implementation of the parallel evaluation of the neighborhood (iteration-level) model for a single LS on GPU. That is the reason why, a natural way for designing multi-start LSs on GPU based on the iteration-level is to iterate the whole process (i.e. the execution of a single LS on GPU) to deal with as many LSs as needed (see Fig. 1). Indeed, in general, evaluating a fitness function for each neighbor is frequently the most costly operation of the LS. Therefore, in this scheme, task distribution is clearly defined: the CPU manages the whole sequential LS process for each LS algorithm and the GPU is dedicated only to the parallel evaluation of solutions.

Algorithm 2 gives the template of this model. The reader is referred to [2,3] for more details about the original algorithm. Basically, for each LS, the CPU first sends the number of expected neighbors to be evaluated to the GPU and then these solutions are processed on GPU. Regarding the kernel thread organization, as quoted above, a GPU is organized following the SPMD model, meaning that each GPU thread associated with one neighbor executes the same evaluation function kernel. Finally, results of the evaluation function are returned back to the host via the global memory.

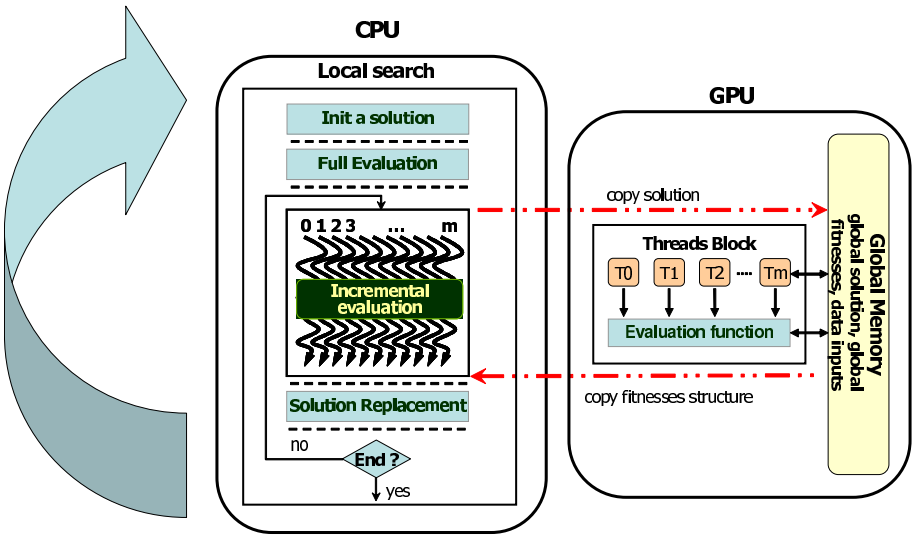

Fig. 1. Multi-start LS algorithms based on the parallel evaluation of the neighborhood on GPU (iteration-level). In this scheme, one thread is associated with one neighbor.

Algorithm 1 Multi-start local search algorithms template on GPU based on the iteration-level model

```
1: Allocate problem data inputs on GPU memory
2: Copy problem data inputs on GPU memory
3: Allocate a solution on GPU memory
4: Allocate a neighborhood fitnesses structure on GPU memory
5: Allocate additional solution structures on GPU memory
6: for m = 1 to #local_searches do
7:    Choose an initial solution
8:    Evaluate the solution
9:    Specific LS initializations
10: end for
11: repeat
12:    for m = 1 to #local_searches do
13:       Copy the solution on GPU memory
14:       Copy additional solution structures on GPU memory
15:       for each neighbor in parallel on GPU do
16:          Incremental evaluation of the candidate solution
17:          Insert the resulting fitness into the neighborhood fitnesses structure
18:       end for
19:       Copy back the neighborhood fitnesses structure on CPU memory
20:       Specific LS solution selection strategy on the neighborhood fitnesses structure
21:       Specific LS post-treatment
22:    end for
23:    Possible cooperation between the different solutions
24: until a stopping criterion satisfied
```

This way, the GPU is used as a coprocessor in a synchronous manner. The time-consuming part i.e. the incremental evaluation kernel is calculated by the GPU and the rest is handled by the CPU. The advantage of this scheme resides in its highly parallel structure (i.e. an important number of generated neighbors to handle), leading to a significant multiprocessors occupancy of the GPU. However, depending on the number of LS algorithms, the main drawback of this scheme is that copying operations from the CPU to the GPU can become frequent and thus can lead to a significant performance decrease.

3.2 Design of Multi-start Local Search Algorithms Based on the Algorithmic-level

A natural way for designing multi-start LSs on GPU is to parallelize the whole LS process on GPU by associating one GPU thread with one LS. This way, the main advantage of this approach is to minimize the data transfers between the host CPU memory and the GPU. Figure 2 illustrates this idea of this full distribution (algorithmic-level). In the rest of this paper, we will focus on this approach.

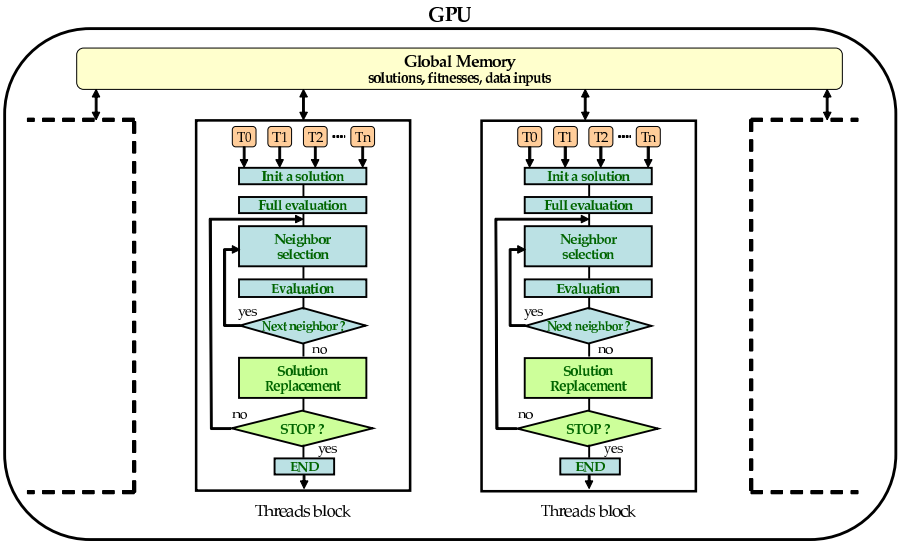

Fig. 2. Multi-start LS algorithms based on the full distribution of LSs on GPU (algorithmic-level). One thread is associated with one local search.

The details of the algorithm are given in Algorithm 1. First of all, at initialization stage, memory allocations on GPU are made: data inputs of the problem must be allocated and copied on GPU (lines 1 and 2). It is important to notice that problem data inputs (e.g. a matrix in the traveling salesman problem [12])

are a read-only structure and never change during all the execution of LS algorithms. Therefore, their associated memory is copied only once during all the execution. Second, a certain number of solutions corresponding to each LS must be allocated on GPU (line 3). Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of incremental evaluation (line 4). Third, during the initialization of the different LS algorithms on GPU, each solution is generated and evaluated (from lines 5 to 9). Fourth, comes the algorithmic-level, in which the iteration process of each LS is performed in parallel on GPU (from lines 11 to 17). Since each neighbor is evaluated in a sequential manner on GPU, unlike the iteration-level scheme, there is no need to allocate and manipulate any neighborhood fitness structure. Fifth, an exchange of the best-so-far solutions could be made to accelerate the search process (line 18). In that case, operations on the global memory may be considered. Finally, the process is repeated until a stopping criterion is satisfied.

Algorithm 2 Multi-start local search algorithms template on GPU based on the algorithmic-level model

```
1: Allocate problem data inputs on GPU memory
2: Copy problem data inputs on GPU memory
3: Allocate #local_searches solutions on GPU memory
4: Allocate #local_searches additional solution structures on GPU memory
5: for each LS in parallel on GPU do
6:     Choose an initial solution
7:     Evaluate the solution
8:     Specific LS initializations
9: end for
10: repeat
11:     for each LS in parallel on GPU do
12:         for each neighbor do
13:             Incremental evaluation of the candidate solution
14:             Specific LS solution selection strategy
15:         end for
16:         Specific LS post-treatment
17:     end for
18:     Possible cooperation between the different solutions
19: until a stopping criterion satisfied
```

3.3 Memory Management of Multi-start Local Search Algorithms on the Algorithmic-level

Memory Coalescing Issues. When an application is executed on GPU, each block of threads is split into SIMD groups of threads called *warps*. At any clock cycle, each processor of the multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data. Global memory is

conceptually organized into a sequence of 128-byte segments. The number of memory transactions performed for a half-warp will be the number of segments having the same addresses than those used by that half-warp. Fig. 3 illustrates an example of the memory management layer for a simple vector addition.

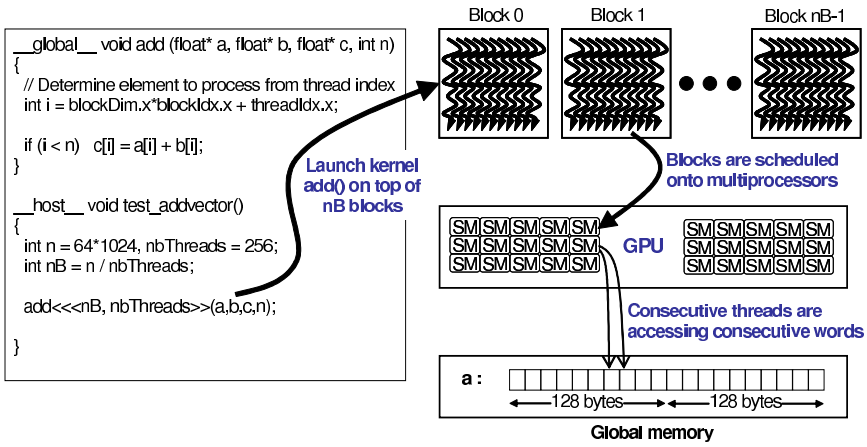

Fig. 3. An example of kernel execution for vector addition.

For more efficiency, global memory accesses must be coalesced, which means that a memory request performed by consecutive threads in a half-warp is associated with precisely one segment. The requirement is that threads of the same warp must read global memory in an ordered pattern. If per-thread memory accesses for a single half-warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. In the example of vector addition, memory accesses to the vectors a and b are fully coalesced, since threads with consecutive thread indices access contiguous words.

Otherwise, accessing scattered locations results in memory divergence and requires the processor to perform one memory transaction per thread. The performance penalty for non-coalesced memory accesses varies according to the size of the data structure. Regarding LS structures, coalescing is difficult when global memory accesses have a data-dependent unstructured pattern (especially for a permutation representation). As a result, non-coalesced memory accesses imply many memory transactions and it can lead to a significant performance decrease for LS methods.

Memory Organization. Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. For instance, the use of texture memory is a solution for reducing memory transactions due to non-coalesced accesses. Texture memory provides a surprising aggregation of capabilities including the ability

to cache global memory (separate from register, global, and shared memory). Regarding the data management on the different GPU memories, the following observations can be made whatever the used multi-start LS algorithm:

- **Global memory:** For each running LS on GPU (one thread), its associated solution is stored on the global memory. The same goes on for additional solution structures. This way, it ensures a global visibility among the different threads (LSs) during the entire search process for a possible cooperation. In a general way, all the data in combinatorial problems could be also associated with the global memory. However, as previously said, non-coalesced memory accesses may lead to a performance decrease. Therefore, the texture memory might be preferred since it can be seen as a relaxed mechanism for the threads to access the global memory. Indeed, the coalescing requirements do not apply to texture memory accesses.
- **Texture memory:** This read-only memory is adapted to LS algorithms since the problem inputs do not change during the execution of the algorithm. In most of optimization problems, problem inputs do not often require a large amount of allocated space memory. As a consequence, these structures can take advantage of the 8KB cache per multiprocessor of texture units. Indeed, minimizing the number of times that data goes through cache can increase significantly the efficiency of algorithms [13]. Moreover, cached texture data is laid out to give best performance for structures with 1D/2D access patterns such as matrices. The use of textures in place of global memory accesses is a completely mechanical transformation. Details of texture coordinate clamping and filtering is given in [10, 14].
- **Constant memory:** This memory is read only from kernels and is hardware optimized for the case where all threads read the same location. It might be used when the calculation of the evaluation function requires a common lookup table for all solutions (e.g. a decoder table for an indirect encoding on the job shop scheduling problem [15]).
- **Shared memory:** The shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. Since this memory area provides a way for threads to communicate within the same block, it might be used with the global memory in the context of a possible cooperation between different LS algorithms. In the case of the multi-start LS model, the type of shared information is the best-so-far solution found at each iteration of the search process.
- **Registers:** Among streaming processors, they are partitioned among the threads running on it and they constitute fast access memory. In the kernel code, each declared variable is automatically put into registers.
- **Local memory:** In a general way, additional structures such as declared array will reside in local memory. In fact, local memory resides in the global memory allocated by the compiler and its visibility is local to a thread (a LS).

Table 1 summarizes the kernel memory management in accordance with the different LS components. For the management of random numbers in SA, effi-

Table 1. Kernel memory management. Summary of the different memories used in the multi-start LS algorithms on GPU.

Type of memory	**LS structure**
Texture memory	problem data inputs
Global memory	candidate solutions, additional candidate solution structures
Shared memory	possible solutions to exchange
Registers	additional LS variables
Local memory	additional LS structures
Constant memory	additional problem lookup tables

cient techniques are provided in many books such as [16] to implement random generators on GPU. For deterministic multi-start LSs based on HC or TS, the random initialization of solutions might be done on CPU and then they can be copied on the GPU via the global memory to perform the LS process. This way, it ensures that the obtained results are the same as a multi-start LS performed on a traditional CPU. Regarding the management of the tabu list on GPU, since the list is particular to a TS execution, a natural mapping is to associate a tabu list to the local memory. However, since this memory has a limited size, large tabu lists should be associated with the global memory instead.

4 Experiments

To validate our approach, the multi-start model has been implemented on the quadratic assignment problem (QAP) on GPU using CUDA. The QAP 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 QAP is equivalent to finding a permutation $\pi = (1, 2, \ldots, n)$ 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 chosen neighborhood for all the experiments is based on a 2-exchange operator ($\frac{n \times (n-1)}{2}$ neighbors). The incremental evaluation function has a time complexity of $O(n)$. The considered instances are the Taillard instances proposed in [17]. They are uniformly generated and are well-known for their difficulty.

The used configuration is an Intel Xeon 3GHz 2 cores with a GTX 280 (30 multiprocessors). From an implementation point of view, to build the CPU test code, the g++ compiler has been used with the -O2 optimization flag and SSE instructions. The specific parameters for each single LS algorithm are given in Table 2.

Table 2. Used parameters for each particular LS.

Tabu search	Simulated annealing
tabu list size: $tl = \frac{n \times (n-1)}{16}$	geometric cooling schedule initial temperature: $T_0 = 10000$
iterations: $iters = 10000$	threshold: $thr = 1$ ratio: $r = 0.9$ iterations: $iters = \frac{n \times (n-1)}{2}$ equilibrium state: $T < thr$

4.1 Measures of the Efficiency of Multi-start Algorithms Based on the Algorithmic-level

In the next experiments, the effectiveness in terms of quality of solutions is not addressed here. Only execution times and acceleration factors are reported in comparison with a mono-core CPU. The objective is to evaluate the impact of a GPU implementation of multi-start algorithms based on the algorithmic-level (i.e. the full distribution of the search process on GPU) in terms of efficiency. For each multi-start algorithm, a standalone mono-core CPU implementation, a pure GPU one, and a GPU version using texture memory (GPU_{tex}) are considered. The number of LS algorithms of the multi-start model is set to 4096 which corresponds to a realistic scenario in accordance with the algorithm convergence. The average time has been measured in seconds for 30 runs. The standard deviation is not represented since its value is very low for each measured instance. The obtained results are reported in Table 3 for the different LS multi-start algorithms on GPU.

Table 3. Measures of the efficiency of the algorithmic-level on the QAP. The average time is reported in seconds for 30 executions, the number of LSs is fixed to 4096.

	tai30a	tai40a	tai50a	tai60a	tai80a	tai100a
HC CPU	5.48	17.18	44.56	88.32	302.43	810.39
HC GPU	$3.19_{\times 1.7}$	$7.44_{\times 2.3}$	$15.79_{\times 2.8}$	$30.06_{\times 2.9}$	$90.45_{\times 3.3}$	$224.51_{\times 3.6}$
HC GPUTex	$1.02_{\times 5.4}$	$2.96_{\times 5.8}$	$6.69_{\times 6.7}$	$12.52_{\times 7.1}$	$41.65_{\times 7.3}$	$103.59_{\times 7.8}$
TS CPU	335.57	725.39	1539.60	2439.86	6097.61	13004.76
TS GPU	$105.12_{\times 3.2}$	$207.12_{\times 3.5}$	$414.50_{\times 3.7}$	$655.32_{\times 3.7}$	$1544.32_{\times 3.9}$	$3222.01_{\times 4.0}$
TS GPUTex	$55.12_{\times 6.1}$	$105.65_{\times 6.9}$	$176.29_{\times 8.7}$	$262.31_{\times 9.3}$	$588.33_{\times 10.4}$	$1207.77_{\times 10.8}$
SA CPU	412.64	874.44	1672.63	2699.89	6807.88	13960.69
SA GPU	$115.32_{\times 3.6}$	$223.65_{\times 3.9}$	$422.32_{\times 4.0}$	$677.28_{\times 4.0}$	$1578.21_{\times 4.3}$	$3121.28_{\times 4.5}$
SA GPUTex	$72.25_{\times 5.7}$	$135.21_{\times 6.5}$	$205.74_{\times 8.1}$	$278.88_{\times 9.7}$	$609.78_{\times 11.2}$	$1161.52_{\times 12.0}$

Regarding the acceleration for a pure implementation on GPU based on HC (HC GPU), it varies between $\times 1.7$ for the instance tai30a to $\times 3.6$ for the last instance. In comparison with a pure CPU implementation, the obtained accel-

eration factors are positive but not impressive. Indeed, due to high misaligned accesses to global memories (flows and distances in QAP), non-coalescing memory reduces the performance of the implementation. Binding texture on global memory allows to overcome the problem (HC GPUTex). Indeed, from the instance tai30a, using texture memory starts providing significant acceleration factors ($\times$5.4). GPU keeps accelerating the LS process as long as the size grows and the best results are obtained for the instance tai100a ($\times$7.8).

Regarding the performance for the other multi-start algorithms (TS and SA based), similar observations can be made. Indeed, on the hand, the obtained speed-ups for the texture version of multi-start algorithms based on TS vary between $\times$6.1 to $\times$10.8. And on the other hand, they vary from $\times$5.7 to $\times$12.0 for the multi-start algorithms based on SA. In a general manner, the performance variation obtained with the different algorithms on GPU is in accordance with the algorithm complexity ($Complexity(SA) >= Complexity(TS) >> Complexity(HC)$).

The point to highlight in these experiments is that organizing data into cache such as texture memory clearly allows to improve the speed-ups in comparison with a standard GPU version where inputs are stored in the global memory.

4.2 Measures of the Efficiency of Large GPU-based Implementations

Another experiment consists in measuring the impact in terms of efficiency by varying the number of LSs in the multi-start based on the algorithmic-level. In addition, we propose to compare this approach with the multi-start based on the iteration-level model (parallel evaluation of the neighborhood on GPU) presented in Section 3.1. For doing this, we propose to deal with the instance tai50a with the same parameters used before in the context of multi-start methods based on TS. The obtained results are depicted in Fig. 4 for the texture optimization.

For the algorithmic-level, one can notice that it starts providing a positive acceleration of $\times$1.7 from a number of 512 LSs (one thread per LS). From 1024 LSs, the acceleration factors are drastically improved until reaching $\times$8.7 for 4096 LSs. After that, the speed-up keeps improving slowly with the size increase. However, as one can see in Fig. 5, no significant difference can be made in terms of the quality of the solutions obtained for more than 168384 LSs. Therefore, since the execution is already time-consuming, it might not be relevant to perform more LSs.

Regarding a small number of running LSs, from 1 to 256 LSs, the multi-start for the algorithmic-level is clearly inefficient. This can be explained by the fact that since the number of threads is relatively small, the number of threads per block is not enough to fully cover the memory access latency.

Unlike the previous model, for the multi-start based on the iteration-level, the obtained speed-ups are quiet regular (from $\times$4.4 to $\times$5.1) whatever the number of running LSs. Indeed, since one thread is associated with one neighbor ($\frac{n\times(n-1)}{2}$ neighbors), during the kernel execution, there is enough threads to keep the GPU multiprocessors busy. However, as one can notice, the maximal performance of

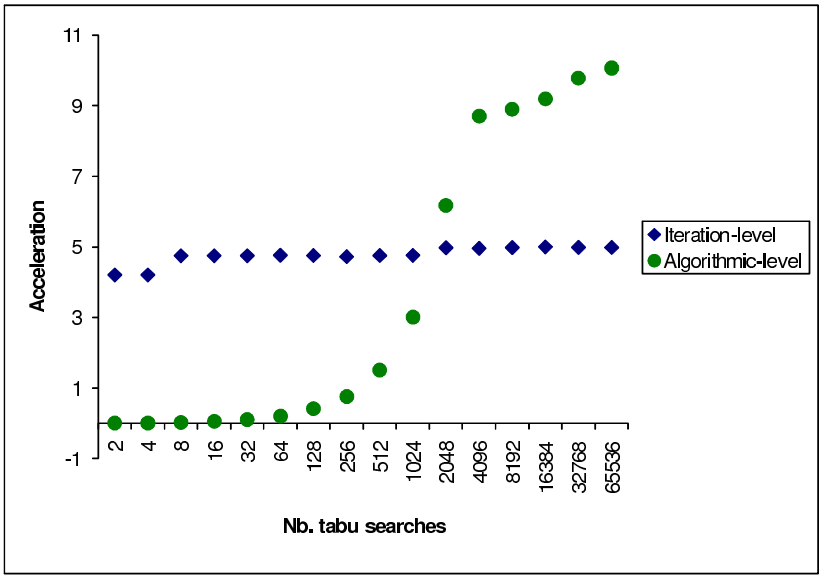

Fig. 4. Measures of the efficiency of the two multi-start approaches using the texture memory algorithmic-level approach in comparison with the iteration-level by varying the number of tabu searches (instance tai50a).

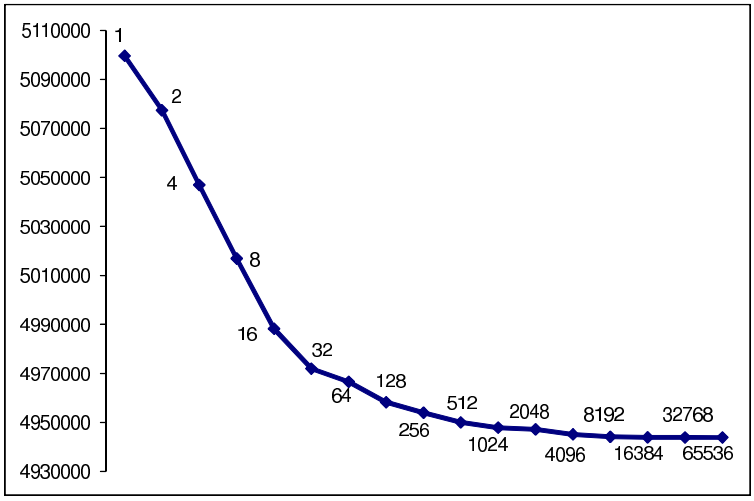

Fig. 5. Measures of the quality of the solutions for the multi-start model based on the algorithmic-level (tai50a). The average fitness is reported for 30 executions where each point represents a certain number of LSs.

this scheme is quiet limited because of the multiple data copies between the CPU and the GPU (see [2] for an analysis of data transfers).

5 Discussion and Conclusion

Parallel metaheuristics such as the multi-start model allow to improve the effectiveness and robustness in optimization problems. Their exploitation for solving real-world problems is possible only by using a great computational power. High-performance computing based on GPU accelerators is recently revealed as an efficient way to use the huge amount of resources at disposal. However, the exploitation of the multi-start model is not trivial and many issues related to the context execution and to the memory hierarchy of this architecture have to be considered.

In this paper, we have proposed a guideline to design and implement general GPU-based multi-start LS algorithms. The different concepts addressed throughout this paper takes into account popular LS algorithms such as HC, SA or TS. The designed and implemented approaches have been experimentally validated on a combinatorial optimization problem. To the best of our best of knowledge, multi-start parallel LS approaches have never been widely investigated so far.

The idea of our methodology is based on two natural schemes which exploit the GPU in a different manner. In the first scheme, the multi-start model is combined with the parallel evaluation of the neighborhood. The advantage of this scheme is to maximize the GPU in terms of multiprocessor occupancy. However, the performance of this scheme is limited due to the data transfers between the CPU and the GPU. To deal with this issue, we have particularly focused on the full distribution of the search process on GPU with the appropriate use of memory. Applying such mechanism with an efficient memory management allows to provide significant speed-ups (up to $\times 12$). However, this second scheme could also present some performance limitations when dealing with a small number of LS executions.

In a general manner, the two proposed schemes are complementary and their use strongly depends of the number of LSs to deal with. It would be interesting to test the performance of our approaches with some combinatorial optimization problems involving the use of different memories such as the constant and the shared memory.

Another perspective of this work is to combine the multi-start on GPU with a pure multi-core approach. Indeed, since this model has a high degree of parallelism, the CPU cores can also work in parallel in an independent manner. Moreover, since nowadays the actual configurations have 4 and 8 cores, instead of waiting the results back from the GPU, this computational power should be well-exploited in parallel to provide additional accelerations.

References

1. Talbi, E.G.: Metaheuristics: From design to implementation. Wiley (2009)

2. Luong, T.V., Melab, N., Talbi, E.G.: Local search algorithms on graphics processing units. a case study: the permutation perceptron problem. In: EvoCOP. Volume 6022 of LNCS., Springer (2010) 264–275
3. Luong, T.V., Melab, N., Talbi, E.G.: Large neighborhood for local search algorithms. In: IPDPS, IEEE Computer Society (2010)
4. Zhu, W., Curry, J., Marquez, A.: Simd tabu search with graphics hardware acceleration on the quadratic assignment problem. International Journal of Production Research (2008)
5. Janiak, A., Janiak, W.A., Lichtenstein, M.: Tabu search on gpu. J. UCS **14**(14) (2008) 2416–2426
6. Alba, E., Talbi, E.G., Luque, G., Melab, N.: 4. Metaheuristics and Parallelism. In: Parallel Metaheuristics: A New Class of Algorithms. Wiley (2005) 79–104
7. Zomaya, A.Y., Patterson, D., Olariu, S.: Sequential and parallel meta-heuristics for solving the single row routing problem. Cluster Computing **7**(2) (2004) 123–139
8. Melab, N., Cahon, S., Talbi, E.G.: Grid computing for parallel bioinspired algorithms. J. Parallel Distributed Computing **66**(8) (2006) 1052–1061
9. 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 Distribributed Computing **68**(10) (2008) 1389–1401
10. NVIDIA: CUDA Programming Guide Version 3.0. (2010)
11. Group, K.: OpenCL 1.0 Quick Reference Card. (2010)
12. Burkard, R.E., Deineko, V.G., Woeginger, G.J.: The travelling salesman problem on permuted monge matrices. J. Comb. Optim. **2**(4) (1998) 333–350
13. Bader, D.A., Sachdeva, V.: A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In Oudshoorn, M.J., Rajasekaran, S., eds.: ISCA PDCS, ISCA (2005) 41–48
14. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda. ACM Queue **6**(2) (2008) 40–53
15. Dell'Amico, M., Trubian, M.: Applying tabu search to the job-shop scheduling problem. Ann. Oper. Res. **41**(1-4) (1993) 231–252
16. NVIDIA: GPU Gems 3. Chapter 37: Efficient Random Number Generation and Application Using CUDA. (2010)
17. Taillard, É.D.: Robust taboo search for the quadratic assignment problem. Parallel Computing **17**(4-5) (1991) 443–455