Large Neighborhood Local Search Optimization on Graphics Processing Units

Thé Van Luong, Nouredine Melab, El-Ghazali Talbi
Dolphin / Opac Team
INRIA Lille Nord Europe / CNRS LIFL
Parc Scientifique de la Haute Borne
40, avenue Halley, Bat. A, Park Plaza
59650 Villeneuve d'Ascq, France
The-Van.Luong@inria.fr, Nouredine.Melab@lifl.fr, El-Ghazali.Talbi@lifl.fr

***Abstract*—Local search (LS) algorithms are among the most powerful techniques for solving computationally hard problems in combinatorial optimization. These algorithms could be viewed as "walks through neighborhoods" where the walks are performed by iterative procedures that allow to move from a solution to another one in the solution space. In these heuristics, designing operators to explore large promising regions of the search space may improve the quality of the obtained solutions at the expense of a highly computationally process. Therefore, the use of graphics processing units (GPUs) provides an efficient complementary way to speed up the search. However, designing applications on GPU is still complex and many issues have to be faced. We provide a methodology to design and implement large neighborhood LS algorithms on GPU. The work has been experimented for binary problems by deploying multiple neighborhood structures. The obtained results are convincing both in terms of efficiency, quality and robustness of the provided solutions at run time.**

***Keywords*-GPU-based metaheuristics; parallel local search algorithms on GPU;**

I. Introduction

Plenty of hard problems in a wide range of areas including engineering design, telecommunications, logistics, biology, etc., have been modeled and tackled successfully with optimization approaches such as metaheuristics (generic heuristics). Local search algorithms are a class of metaheuristics which handle with a single solution iteratively improved by exploring its neighborhood in the solution space. Fig. 1 gives a general model for LS algorithms. At each iteration, a set of neighboring solutions is generated and evaluated. The best of these candidate solutions is selected to replace the current solution. The process is iterated until a stopping criterion is satisfied.

Common LS heuristics of the literature are hill climbing, simulated annealing, tabu search, iterative local search and variable neighborhood search. A state-of-the-art of LS algorithms can be found in [1].

The definition of the neighborhood is a required common step for the design of any LS algorithm. The neighborhood structure plays a crucial role in the performance of a LS method. Theoretical and experimental studies have shown that the increase of the neighborhood size may improve the effectiveness (quality of provided solutions) of the LS algorithms [2]. Nevertheless, as it is generally CPU time-consuming it is not often fully exploited in practice. Indeed, experiments with large neighborhood algorithms are often stopped without convergence being reached. That is the reason why, in designing LS methods, there is often a compromise between the size of the neighborhood to use and the computational complexity to explore it. As a consequence, in LS algorithms, there is often a reduction of the size of the explored neighborhood at the expense of the effectiveness. To deal with such issues, only the use of parallelism allows to design algorithms based on large neighborhoods.

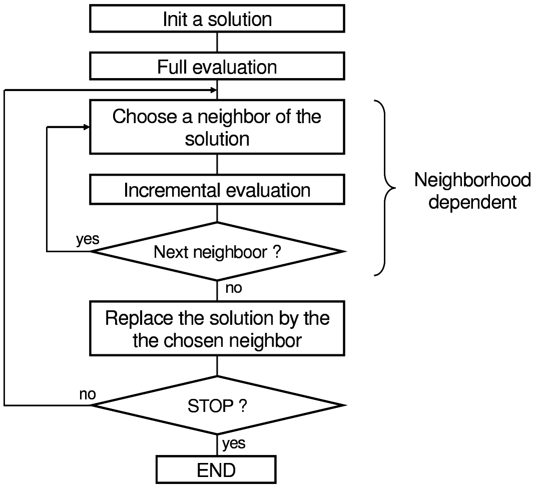


Figure 1. General model for local search algorithms

Nowadays, GPU computing is recognized as a powerful way to achieve high-performance on long-running scientific applications [3]. Designing LS algorithms based on large neighborhood structures for solving real-world optimization problems are good challenges for GPU computing. However, to the best of our knowledge only few research works related to evolutionary algorithms on GPU exist [4]–[7]. Indeed, the parallel exploration of the neighborhood on GPU is not immediate and several challenges persist and are particular related to the characteristics and underlined issues of the GPU architecture and the LS algorithms.

In this paper, we contribute with the first results of LS algorithms based on large neighborhoods on GPU. Finding an efficient task distribution of the LS process onto the GPU organization is a challenging issue dealt with in this paper. In other words, one has to identify what must be performed on CPU and GPU. Another handled issue is to find correct mappings between the different neighborhood structures and the hierarchical GPU. Since the neighborhood structure strongly depends on the target optimization problem, we focus on binary problems all along of this paper. We propose to deal with three neighborhoods of different sizes.

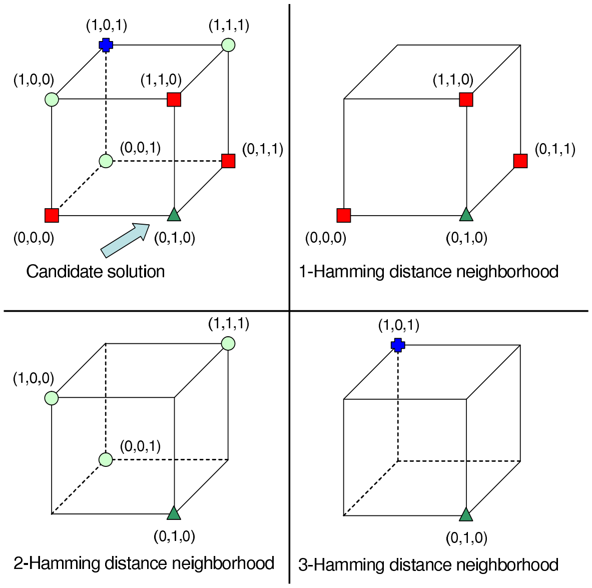
To be valided the work has been experimented on the permuted perceptron problem (PPP) introduced by Pointcheval [8]. The problem is a cryptographic identification scheme based on NP-complete problems, which seems to be well suited for resource constrained devices such as smart cards. The work has been experimented on different popular instances of the literature. We investigate to measure the impact on how the increase of the size of the neighborhood can improve the quality of the obtained solutions.

The rest of the paper is organized as follows: Section II presents the three handled neighborhoods for binary problems. In Section III, a generic LS algorithm and efficient mappings on GPU are proposed for each neighborhood structure. Application of this methodology is made for the permuted perceptron problem in Section IV. Finally, conclusions and a discussion of this work are drawn in Section V.

II. Neighborhoods for Binary Problems

Designing any iterative metaheuristic needs an encoding of a solution. The encoding must be suitable and relevant to the tackled optimization problem. For binary problems, any candidate solution is represented by a vector (or string) of binary values. Moreover, the efficiency of a representation is related to the search operators applied on this representation i.e. the neighborhood.

The natural neighborhood for binary representations is based on the Hamming distance. This distance measures the number of positions between two strings of equal length in which the corresponding symbols are different. Fig. 2 gives an illustration of the Hamming distance for strings of length 3. For instance, the Hamming distance between the node $(0, 1, 0)$ (represented by a triangle) and each node represented by a circle is equal to two. Therefore, nodes of a same shape in the graph constitute a particular neighborhood of the node $(0, 1, 0)$.



Candidate solution

1-Hamming distance neighborhood

2-Hamming distance neighborhood

3-Hamming distance neighborhood

Figure 2. Three neighborhoods for binary problems

The natural neighborhood for binary representations is based on the Hamming distance. This distance measures the number of positions between two strings of equal length in which the corresponding symbols are different.

- *1-Hamming Distance Neighborhood*. In most cases, the associated neighborhood for binary representations is based on the Hamming distance equal to one. In this neighborhood, generate a neighbor consists in flipping one bit of the candidate vector solution (see Fig. 3). Considering a candidate vector solution of size n, the size of the associated neighborhood is n.
- *2-Hamming Distance Neighborhood*. For binary problems, an improved neighborhood for LS algorithms is based on the Hamming distance of two. It consists on building a neighbor by flipping two values of a candidate solution vector. Two indexes represent a particular neighbor. For a candidate solution of size n, the number of neighbors is $\frac{n \times (n-1)}{2}$. Fig. 4 gives an illustration of this neighborhood.
- *3-Hamming Distance Neighborhood*. An instance of a large neighborhood is a neighborhood built by modifying three values called 3-Hamming distance neighborhood. This neighborhood is much com-

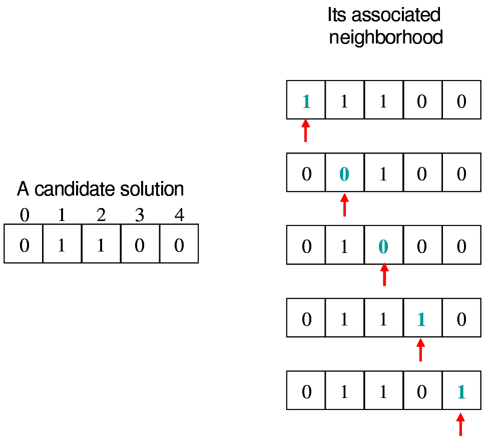


Figure 3. 1-Hamming Distance Neighborhood

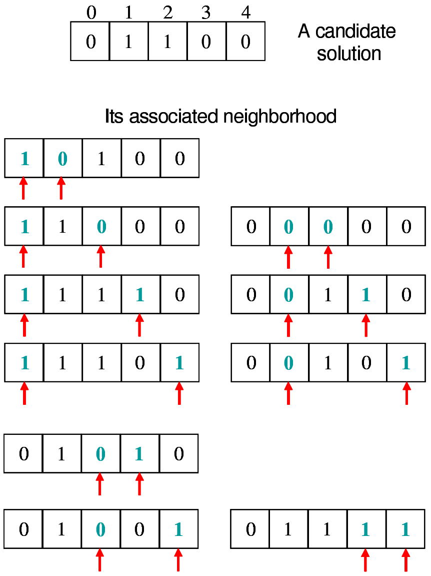


Figure 4. 2-Hamming distance neighborhood

plex since each neighboring solution is identified by 3 indexes (see Fig. 5). The number of elements associated to this neighborhood is $\frac{n\times(n-1)\times(n-2)}{6}$.

Most of the LS algorithms use neighborhoods which are in general a linear (e.g. 1-Hamming distance) or quadratic (e.g. 2-Hamming distance) function of the input instance size. Some large neighborhoods may be high-order polynomial of the size of the input instance (e.g. 3-Hamming distance). Then, the complexity of the search will be much higher. So, in practice, large neighborhoods algorithms are unusable because of their high computational cost. In the other sections, we will show how the use of GPU computing allows to fully exploit parallelism in such algorithms.

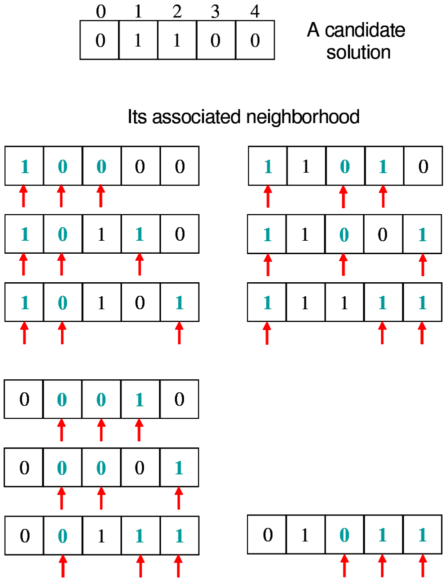


Figure 5. 3-Hamming distance neighborhood

III. Design of Large Neighborhood Local Search Algorithms on GPU

GPU computing may be used to speed-up the search process when designing large neighborhood algorithms. However, the use of GPU-based parallel computing for metaheuristics is not straightforward. Indeed, several challenges mainly related to the hierarchical memory management have to be considered: (1) the distribution of the search process among the CPU and the GPU minimizing the data transfer between them; (2) finding the efficient mapping of the neighborhood structure on the hierarchical GPU. We propose in this section to deal with such issues.

A. GPU Kernel Execution Model

Each processor device on GPU supports the single program multiple data (SPMD) model, i.e. multiple autonomous 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 simultaneously by several processors in parallel.

This kernel handling is dependent of the general-purpose language. For instance, CUDA and OpenCL are parallel computing environments, which provide an application programming interface for NVIDIA architectures [9]. The concept of GPU thread does not have exactly the same meaning as CPU thread. A thread on GPU can be seen as an element of the data to be processed. Compared to CPU threads, GPU threads are lightweight. That means that changing the context between two threads is not a costly operation.

Regarding their spatial organization, threads are organized within so called thread blocks. A kernel is executed by multiple equally threaded blocks. 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. Thus, a unique *id* can be deduced for each thread to perform computation on different data.

B. *The Proposed GPU-based Algorithm*

Adapting traditional LS methods to GPU is not a straightforward task because hierarchical memory management on GPU has to be handled. Indeed, memory transfers from CPU to GPU are slow and these copying operations have to be minimized. We propose a methodology to adapt LS methods on GPU in a generic way. Task distribution is clearly defined: the CPU manages the whole sequential LS process and the GPU is dedicated to the costly part i.e. the parallel generation and evaluation of the neighboring solutions. Algorithm 1 gives a template of any local search algorithms on GPU.

First of all, at initialization stage, memory allocations on GPU are made: data inputs and candidate solution of the problem must be allocated (lines 4 and 5). Since GPUs require massive computations with predictable memory accesses, a structure has to be allocated for storing all the neighborhood fitnesses at different addresses (line 6). Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of incremental evaluation (line 7). Second, problem data inputs, initial candidate solution and additional structures associated to this solution have to be copied on the GPU (lines 8 to 10). It is important to notice that problem data inputs 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. Third, then comes the parallel evaluation of the neighborhood (GPU kernel), in which each neighboring solution is generated and evaluated (from lines 12 to 15). The results of the evaluated solutions (fitnesses) are stored into a data structure. Fourth, since the order in which candidate neighbors are evaluated is undefined, the previous neighborhood fitnesses structure has to be copied to the host CPU (line 16). Then a specific LS solution selection strategy is applied to this structure (line 17): the exploration of the neighborhood fitnesses structure is done by the CPU in a sequential way. Finally, after a new candidate has been selected, this latter and its additional associated structures are copied to the GPU (lines 19 and 20). The process is repeated until a stopping criterion is satisfied.

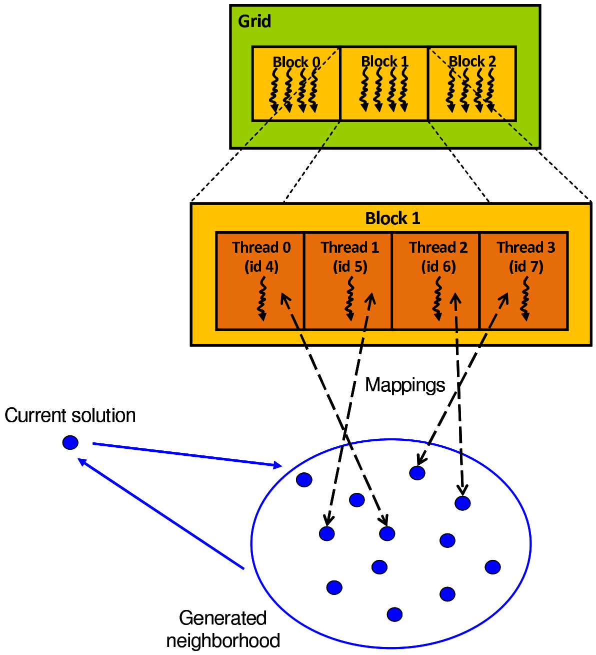


Figure 6. Mappings between threads and neighbors

C. *Efficient Mappings of Neighborhood Structures*

The remaining critical issue is to find efficient mappings between a GPU thread and a particular neighbor. Indeed, this step is crucial in the design of new large neighborhood LS algorithms for binary problems since it is clearly identified as the gateway between a GPU process and a candidate neighbor. As suggested in Fig. 6, on the one hand, the thread *id* is represented by a single index. On the other hand, the move representation of a neighbor varies according to the neighborhood.

1) 1-Hamming Distance: For neighborhoods based on a Hamming distance of one, a mapping between LS neighborhood encoding and GPU threads is quiet direct. Indeed, for a binary vector of size n, the neighborhood size is exactly n where each neighbor is represented by one index varying from 0 to $n-1$. Regarding the GPU threads, they are provided with a unique *id* and thus associated with one single index in a similar manner. That way, the associated kernel can be launched with n threads (each neighbor is associated to a single thread). As a result, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping can be made in constant time.

2) 2-Hamming Distance: For a binary vector of size n, the size of this new neighborhood is $\frac{n\times(n-1)}{2}$. The associated kernel is executed by $\frac{n\times(n-1)}{2}$ threads. For this encoding a mapping between a neighbor and a GPU thread is not straightforward. Indeed, on the one hand, a neighbor is composed by two indexes to modify. On the other hand, threads are identified by a unique *id* (single index). As a result, a $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping has

Algorithm 1 Local Search Template on GPU

```
Choose an initial solution
Evaluate the solution
Specific LS initializations
Allocate problem data inputs on GPU device memory
Allocate a solution on GPU device memory
Allocate a neighborhood fitnesses structure on GPU device memory
Allocate additional solution structures on GPU device memory
Copy problem data inputs on GPU device memory
Copy the solution on GPU device memory
Copy additional solution structures on GPU device memory
repeat
  for each neighbor in parallel on GPU do
    Incremental evaluation of the candidate solution
    Insert the resulting fitness into the neighborhood fitnesses structure
  end for
  Copy neighborhood fitnesses structure on CPU host memory
  Specific LS solution selection strategy on the neighborhood fitnesses structure
  Specific LS post-treatment
  Copy the chosen solution on GPU device memory
  Copy additional solution structures on GPU device memory
until a stopping criterion satisfied
```

to be considered to transform one index into two. In a similar way, a $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping must be handled to transform two indexes into one.

Proposition 1: **Two-to-one index transformation**

Given i and j the indexes of two elements to modify in the binary representation, the corresponding index $f(i,j)$ in the neighborhood representation is equal to $i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2}$, where n is the vector size.

Proposition 2: **One-to-two index transformation**

Given $f(i,j)$ the index of the element in the neighborhood representation, the corresponding index i is equal to $n - 2 - \lfloor \frac{\sqrt{8 \times (m - f(i,j) - 1) + 1} - 1}{2} \rfloor$ and j is equal to $f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1$ in the binary representation, where n is the vector size and m the neighborhood size.

The proofs of one-to-two and two-to-one index transformations can be found in [10]. The complexity of such mappings is a nearly constant time i.e. it depends on the calculation of the square root on GPU (solving quadratic equation).

3) 3-Hamming Distance: For an array of size n, the size of this neighborhood is $\frac{n \times (n-1) \times (n-2)}{6}$. The associated kernel on GPU is executed by $\frac{n \times (n-1) \times (n-2)}{6}$ threads. A mapping here between a neighbor and a GPU thread is also particularly challenging. $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ and $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mappings must be handled efficiently.

The mapping for this neighborhood is a generalization of the 2-Hamming distance neighborhood with a third index. The complexity of the mappings is logarithmic in practice i.e. it depends on the numerical Newton-Raphson method (solving cubic equation).

IV. Application to the Permuted Perceptron Problem

A. Permuted Perceptron Problem

As illustration of a binary problem, the PPP is a NP-complete problem that has received a great attention given its importance in security protocols. An ϵ-vector is a vector with all entries being either +1 or -1. Similarly an ϵ-matrix is a matrix in which all entries are either +1 or -1. The PPP is defined as follows according to [8]:

Definition 1: Given an ϵ-matrix A of size $m \times n$ and a multi-set S of non-negative integers of size m, find an ϵ-vector V of size n such that $\{\{(AV)_j / j = \{1, \ldots, m\}\}\} = S$.

Let $Y = AV$ be a matrix-vector product. Determine a histogram vector H over the integers such that $H_i = \#\{Y_j = i \mid j = 1, \ldots, m\}$. Let V' denote a candidate for the solution V, let $Y' = AV'$ and let H'_i denote the histogram vector of Y'. Then an objective function is given in [11] by:

$$f(V') = 30 \times \sum_{i=1}^{m} (|(AV')_i| - (AV')_i) + \sum_{i=1}^{n} (|H_i - H'_i|).$$

This corresponds to a minimization problem where a value $f(V') = 0$ gives a successful solution to the problem.

B. Configuration

A tabu search [12] has been implemented on CUDA for each neighborhood. This algorithm is an instance of the general LS model presented in introduction. Basically, this algorithm uses a tabu list (a short-term memory) which contains the solutions that have been visited in the recent past. More details of this algorithm are given in [12].

The used configuration is an Intel Core 2 Duo 2.67GHz with a NVIDIA GTX 280 card. The number of multiprocessors of this card is equal to 30 and the constraints of memory alignment are relaxed in comparison with the previous architectures (G80 series). Therefore, GTX 280 cards get better global memory performance.

The following experiments intend to measure the quality of the solutions for the instances of the literature addressed in [11]. A tabu search has been executed 50 times with a maximum number of $\frac{n\times(n-1)\times(n-2)}{6}$ iterations (stopping criterion). The tabu list size has been arbitrary set to a $\frac{m}{6}$ where m is the number of neighbors. The average value of the evaluation function (fitness) and its standard deviation (in sub index) has been measured. The number of successful tries (fitness equal to zero) and the average number of iterations are also represented.

C. 1-Hamming Distance

Table I reports the results for the tabu search based on the 1-Hamming distance neighborhood. In a short execution time, the algorithm has been able to find few solutions for the instances $m = 73, n = 73$ (10 successful tries on 50) and $m = 81, n = 81$ (6 successful tries on 50). The two other instances are well-known for their difficulties and no solutions were found. Regarding execution time, GPU version does not offer anything in terms of efficiency. Indeed, since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency.

To measure the efficiency of the GPU-based implementation of this neighborhood, bigger instances of the PPP must be considered. Fig. 7 shows the GPU acceleration factor for different PPP instance sizes on the base of 10000 iterations.

From $m = 201$ and $n = 217$, the GPU version starts to be faster than CPU version (acceleration factor of $\times 1.1$). As long as the problem size increases, the speed-up grows significantly (up to $\times 10.8$ for $m = 1501$ and $n = 1517$).

D. 2-Hamming Distance

A tabu search has been implemented on GPU using a 2-Hamming distance neighborhood. Results of the experiment for the PPP are reported in Table II.

By using this other neighborhood, in comparison with Table I, the quality of solutions has been significantly improved: on the one side the number of successful tries for both $m = 73, n = 73$ (19 solutions) and $m = 81, n = 81$ (13 solutions) is more important. On the other side, 12 solutions were found for the instance $m = 101, n = 101$. Regarding execution time, acceleration factor for GPU version is really efficient (from $\times 9.9$ to $\times 18.5$). Indeed, since a large number of threads are executed, GPU can take full advantage of the multiprocessors occupancy.

E. 3-Hamming Distance

A tabu search using a 3-Hamming distance neighborhood has been implemented. Since the computational time is too exorbitant, the average expected time for the CPU implementation is deduced from the base of 100 iterations per execution. Results are collected in Table III.

In comparison with Knudsen and Meier article [11], the results found by the generic tabu search are competitive without any use of cryptoanalysis techniques. Indeed, the number of successful solutions has been drastically improved for every instance (respectively 35, 28 and 18 successful tries) and a solution has been even found for the difficult instance $m = 101, n = 117$. Regarding execution time, acceleration factors using GPU are very significant (from $\times 24.2$ to $\times 25.8$).

The conclusions from this experiment indicate that the use of GPU provides an efficient way to deal with large neighborhoods. Indeed, 3 Hamming-distance neighborhood on PPP were unpracticable in terms of single CPU computational resources. So, implementing this algorithm on GPU has allowed to exploit parallelism in such neighborhood and improve the quality of solutions.

V. Discussion and Conclusion

Local search algorithms based on large neighborhoods may allow to enhance the effectiveness in combinatorial optimization [2]. However, their exploitation for solving real-world problems is possible only by using a great computing power. High-performance computing based on GPU accelerators is recently revealed as an efficient way to use the huge amount of resources at disposal and fully exploit the parallelism of neighborhoods. To the best of our knowledge, no research work has been published on LS algorithms on GPU based on different neighborhoods exploration.

Table I
PERMUTED PERCEPTRON PROBLEM 1-HAMMING DISTANCE

Problem	Fitness	# iterations	# solutions	CPU time	GPU time
73 × 73	$10.3_{5.1}$	59184.1	10/50	$4s$	$9s$
81 × 81	$10.8_{5.6}$	77321.3	6/50	$6s$	$13s$
101 × 101	$20.2_{14.1}$	166650	0/50	$16s$	$33s$
101 × 117	$16.4_{5.4}$	260130	0/50	$29s$	$57s$

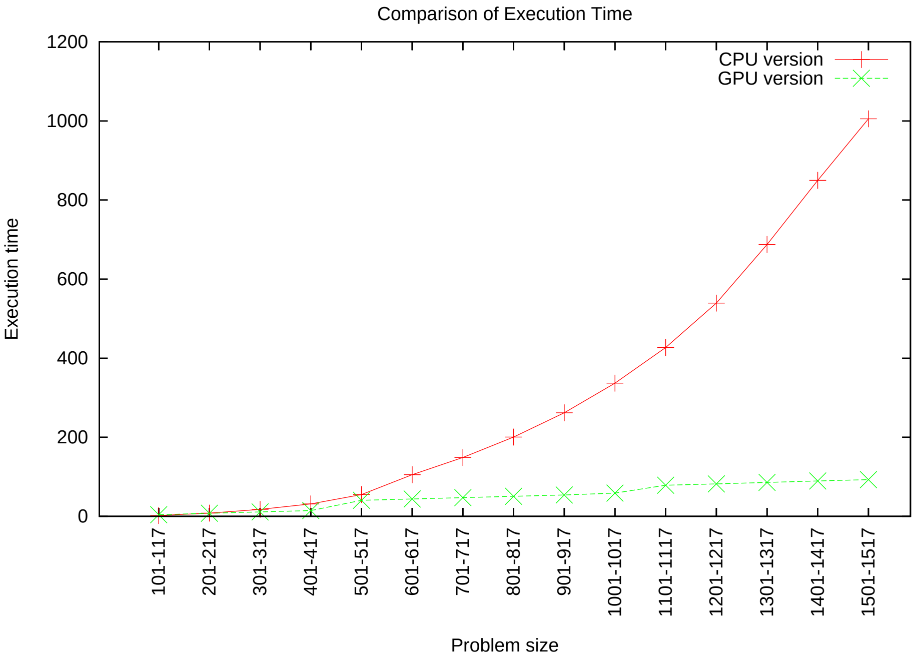


Figure 7. Comparison of the execution time for large PPP instances

Table II
PERMUTED PERCEPTRON PROBLEM 2-HAMMING DISTANCE

Problem	Fitness	# iterations	# solutions	CPU time	GPU time	Acceleration
73 × 73	$16.4_{17.9}$	43031.7	19/50	$81s$	$8s$	×9.9
81 × 81	$15.5_{16.6}$	67462.5	13/50	$174s$	$16s$	×11.0
101 × 101	$14.2_{14.3}$	138349	12/50	$748s$	$44s$	×17.0
101 × 117	$13.8_{10.8}$	260130	0/50	$1947s$	$105s$	×18.5

Table III
PERMUTED PERCEPTRON PROBLEM 3-HAMMING DISTANCE

Problem	Fitness	# iterations	# solutions	CPU expected time	GPU time	Acceleration
73 × 73	$2.4_{4.3}$	21360.2	35/50	$1202s$	$50s$	×24.2
81 × 81	$3.5_{4.4}$	43230.7	28/50	$3730s$	$146s$	×25.5
101 × 101	$6.2_{5.4}$	117422	18/50	$24657s$	$955s$	×25.8
101 × 117	$7.7_{2.7}$	255337	1/50	$88151s$	$3551s$	×24.8

In this paper, a focus has been particularly made on the design of three different neighborhoods to the hierarchical GPU for binary problems. The designed and implemented approaches have been experimentally validated on a cryptographic application. The experiments indicate that GPU computing allows not only to speed up the search process, but also to exploit large neighborhoods structures to improve the quality of the obtained solutions. For instance, LS algorithms based on a Hamming distance of three were unpracticable on traditional machines because of their high computational cost. So, GPU computing has permitted their achievement and the obtained results are particularly promising in terms of effectiveness. Indeed, all along the paper, we have investigated on how the increase of the size of neighborhood allows to improve the quality of the solutions. Furthermore, we strongly believe that the quality of the solutions would be drastically enhanced by (1) increasing the number of running iterations of the algorithm and (2) introducing appropriate cryptonanalysis heuristics.

Beyond the improvement of the effectiveness, the parallelism of GPUs allows to push far the limits in terms of computational resources. As a consequence, a next perspective is to use a multi-GPU approach to allow handling larger neighborhoods. It will consist of partitioning the neighborhood set, where each partition is executed on a single GPU. That way, multi-GPU approach will allow to increase the speed-up of the exploration space of a given solution. But since each GPU has its own private memory, managing the context execution of different GPUs in an efficient way is not a straightforward task.

In the future, GPU concepts will be integrated in the ParadisEO platform. This framework was developped for the design of parallel hybrid metaheuristics dedicated to the mono/multiobjective resolution [13]. ParadisEO can be seen as a white-box object-oriented framework based on a clear conceptual separation of metaheuristics concepts. The Parallel Evolving Objects (PEO) module of ParadisEO includes the well-known parallel and distributed models for metaheuristics. This module will be extended in the future with GPU-based implementation.

References

[1] E.-G. Talbi, *From design to implementation*. Wiley, 2009.

[2] R. K. Ahuja, J. Goodstein, A. Mukherjee, J. B. Orlin, and D. Sharma, “A very large-scale neighborhood search algorithm for the combined through-fleet-assignment model,” *INFORMS Journal on Computing*, vol. 19, no. 3, pp. 416–428, 2007.

[3] 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 Distrib. Comput.*, vol. 68, no. 10, pp. 1389–1401, 2008.

[4] J.-M. Li, X.-J. Wang, R.-S. He, and Z.-X. Chi, “An efficient fine-grained parallel genetic algorithm based on gpu-accelerated,” in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference*, 2007, pp. 855–862. [Online]. Available: http://dx.doi.org/10.1109/NPC.2007.108

[5] D. M. Chitty, “A data parallel approach to genetic programming using programmable graphics hardware,” in *GECCO*, 2007, pp. 1566–1573.

[6] T.-T. Wong and M. L. Wong, “Parallel evolutionary algorithms on consumer-level graphics processing unit,” in *Parallel Evolutionary Computations*, 2006, pp. 133–155.

[7] K.-L. Fok, T.-T. Wong, and M. L. Wong, “Evolutionary computing on consumer graphics hardware,” *IEEE Intelligent Systems*, vol. 22, no. 2, pp. 69–78, 2007.

[8] D. Pointcheval, “A new identification scheme based on the perceptrons problem,” in *EUROCRYPT*, 1995, pp. 319–328.

[9] NVIDIA, *CUDA Programming Guide Version 2.1*, 2009.

[10] T. V. Luong, N. Melab, and E.-G. Talbi, “Parallel Local Search on GPU,” INRIA, Research Report RR-6915, 2009. [Online]. Available: http://hal.inria.fr/inria-00380624/en/

[11] L. R. Knudsen and W. Meier, “Cryptanalysis of an identification scheme based on the permuted perceptron problem,” in *EUROCRYPT*, 1999, pp. 363–374.

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

[13] S. Cahon, N. Melab, and E.-G. Talbi, “Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics,” *J. Heuristics*, vol. 10, no. 3, pp. 357–380, 2004.