Parallel Processing Letters

Neighborhood Structures for GPU-based Local Search Algorithms

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

Opac, CNRS LIFL
Dolphin Project, INRIA Lille Nord Europe
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

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

Local search algorithms are powerful heuristics for solving computationally hard problems in science and industry. In these methods, designing neighborhood operators to explore large promising regions of the search space may improve the quality of the obtained solutions at the expense of a high computation process. As a consequence, the use of GPU computing provides an efficient 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 different neighborhood structures for LS algorithms on GPU. The work has been evaluated for binary problems and 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

1. Introduction

A large number of real-life optimization problems in science, engineering, economics, and business are complex and difficult to solve. Using approximate algorithms such as metaheuristics is the main alternative to solve this class of problems. Local search algorithms (LS) are a class of metaheuristics which manipulate and transform a single solution 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. A complete review of LS algorithms can be found in [1].

In these methods, the definition of the neighborhood plays a crucial role in the performance of the algorithm. Indeed, theoretical and experimental studies have

*Visiting Professor, King Saud University, Riyadh, Saudi Arabia

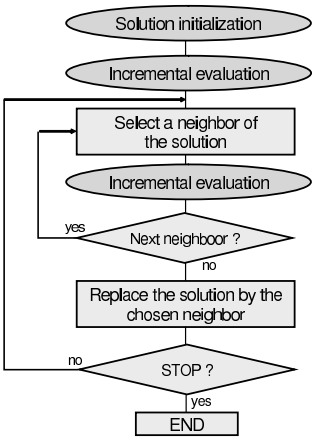

Fig. 1. General model for local search algorithms.

shown that the increase of the neighborhood size may improve the quality of the obtained solutions [2]. Nevertheless, as it is generally CPU time-consuming, this mechanism is not often fully exploited in practice. Indeed, experiments using large neighborhoods are often stopped without convergence being reached. Thereby, in designing LS methods, there is often a trade-off between the size of the neighborhood to use and the computational complexity to explore it. To deal with such issues, only the use of parallelism allows to design methods based on large neighborhood structures.

Nowadays, the calculation on GPU is recognized as an efficient 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 related to the characteristics and underlined issues of the GPU architecture and the LS structures.

The contribution of this work is on the first results of LS algorithms based on different neighborhood structures 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 GPU and manage the data on the different available memories. Another handled issue concerns the adaptation of LS neighborhood structures with the GPU context execution. Since the neighborhood structure strongly depends on the target optimization problem, we focus on binary problems all along of this paper. Thereby, we propose to deal with three neighborhoods of different sizes. More exactly, the focus is on finding efficient/correct mappings between the different neighborhood structures and the GPU thread execution model.

The work has been experimentally validated on the permuted perceptron problem (PPP) [8]. Throughout this paper, we investigate to measure the impact on how the increase of the size of the neighborhood can improve 1) the performance in terms of efficiency and 2) the quality of the obtained solutions.

The rest of the paper is organized as follows: Section 2 presents the three neighborhoods quoted above. In Section 3, generic concepts for designing parallel LS methods on GPU are presented. Section 4 highlights on the main GPU issues that need to be solved to obtain the best performance: a depth look on the GPU memory management and efficient mappings are provided for each neighborhood structure. An application of this methodology and a performance analysis is done for the permuted perceptron problem in Section 5. Finally, conclusions and a discussion of this work are drawn in Section 6.

2. Neighborhood Structures 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 three neighborhoods based on different Hamming distances.

- *1-Hamming Distance Neighborhood*. A standard neighborhood for binary representations is based on the Hamming distance equal to one. In this neighborhood, generating a neighbor consists in flipping one bit of the candidate vector solution. 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 in 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 $\frac{n\times(n-1)}{2}$.
- *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 complex since each neighboring solution is identified by 3 indexes. 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

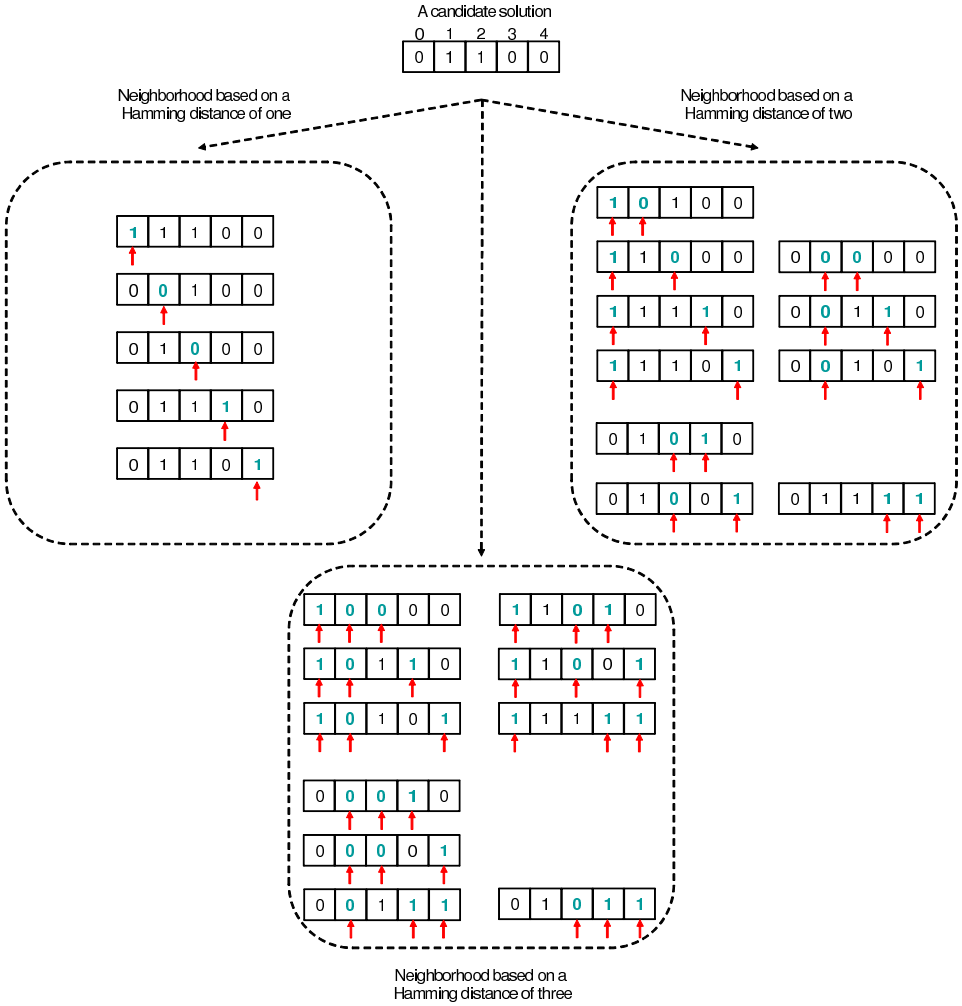

Fig. 2. Neighborhoods based on different Hamming distances.

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.

3. Design of 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, the adaptation of LS algorithms on GPU requires to take into account at the same time the characteristics of the GPU

context execution and the LS structures. In this section, the focus is on the re-design of the general LS model on GPU. The distribution of the search process among the CPU and the GPU has to be faced to minimize the data transfer between them.

3.1. *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 a GPU thread does not have exactly the same meaning as a 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.

3.2. *The Proposed GPU-based Algorithm*

Adapting traditional LS methods to GPU is not a straightforward task because a hierarchical memory management on GPU has to be handled. Indeed, memory transfers from the CPU to the 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. The 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 LS algorithms on GPU.

First of all, at initialization stage, memory allocations on GPU are made: data inputs and the 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 the incremental evaluation (line 7). Second, problem data inputs, the initial candidate solution and some 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

Algorithm 1 Local search template on GPU.

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

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 the 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.

4. GPU Performance Optimization

The task repartition between the CPU and the GPU has been proposed in the previous section through a high-level template for LS algorithms. The next step is to deal with some specific GPU issues to obtain the best performance. Indeed, several challenges mainly related to the GPU context execution have to be considered: (1) the association of the different LS structures with the different available memories; (2) finding the efficient mapping of the neighborhood structure with the GPU threads. We propose in this section to deal with such issues.

4.1. *GPU Kernel Memory Management*

Optimizing the performance of GPU applications often involves optimizing data accesses which includes the appropriate use of the various GPU memory spaces. The use of texture memory is a solution for reducing memory transactions due to non-coalesced accesses. The texture memory provides a surprising aggregation of capabilities including the ability to cache global memory (separate from register, global, and shared memory). Indeed, the texture memory provides an alternative memory access path that can be bound to regions of the global memory. Each texture unit has some internal memory that buffers data from the global memory. Therefore, the texture memory can be seen as a relaxed mechanism for the threads to access the global memory because the coalescing requirements do not apply to texture memory accesses. Thereby, the use of texture memory is well adapted for LS algorithms for the following reasons:

- Data accesses are frequent in the computation of LS incremental evaluation methods. Then, using the texture memory can provide a high performance improvement by reducing the number of memory transactions.
- The texture memory is a read-only memory i.e. no writing operations can be performed on it. This memory is adapted to LS algorithms since the problem data and the solution representation are also read-only values.
- Minimizing the number of times that data goes through cache can increase significantly the efficiency of algorithms [10]. In most of optimization problems, problem inputs (such as data problem or solution representation) 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.
- Cached texture data are laid out to give the best performance for 1D/2D access patterns. The best performance will be achieved when the threads of a warp read locations that are close together from a spatial locality perspective. Since optimization problem inputs are generally 2D matrices or 1D solution vectors, LS structures can be bound to the texture memory.

The use of textures in place of global memory accesses is a completely mechanical transformation. Details of texture coordinate clamping and filtering are given in [9,11]. Table 1 summarizes the kernel memory management in accordance with the different LS structures.

Table 1. Summary of the different memories used in the evaluation function.

Type of memory	**LS structure**
Texture memory	data inputs, solution representation
Global memory	neighborhood fitnesses structure
Registers	additional variables
Local memory	additional structures

The inputs of the problem (e.g. a matrix in TSP) and the solution which generates the neighborhood are associated with the texture memory. The fitnesses structure which stores the obtained results for each neighbor is associated with the global memory. Indeed, since only one writing operation per thread is performed at each iteration, this structure is not part of intensive calculation. Declared variables for the computation of the evaluation function of each neighbor are automatically associated with registers by the compiler. Additional complex structures which are private to a neighbor will reside in local memory. Regarding the shared memory, since it implies local synchronizations in each threads block, its use is not appropriate in the proposed GPU-based LS algorithm.

4.2. *Mappings of Neighborhood Structures on GPU*

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. 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.

4.2.1. *Mapping for a neighborhood based on a 1-Hamming distance*

For neighborhoods based on a Hamming distance of one, a mapping between the 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.

4.2.2. *Mappings for a neighborhood based on a 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 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 4.1. ***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 4.2. ***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 two-to-one and one-to-two index transformations can be found in Appendix A and Appendix B. The complexity of such mappings is in nearly constant time i.e. it depends on the calculation of the square root on GPU (solving quadratic equation).

4.2.3. *Mappings for a neighborhood based on a 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 (see Appendix C and Appendix D). The complexity of the mappings is logarithmic in practice i.e. it depends on the numerical Newton-Raphson method (solving cubic equation).

5. Application to the Permuted Perceptron Problem

5.1. *Definition of the 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. 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. 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 5.1. 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 [12] 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.

5.2. *Configuration for the Experiments*

A tabu search has been implemented on CUDA for each neighborhood using the texture optimization. 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 [13].

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 a better global memory performance.

The following experiments intend to measure the quality of the solutions for the instances of the literature addressed in [12]. 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 $\frac{m}{6}$ where m is the number of neighbors. The average value of the evaluation function (fitness) and its standard deviation (in sub index) have been measured. The number of successful tries (fitness equal to zero) and the average number of iterations are also represented.

5.3. *Neighborhood based on a 1-Hamming Distance*

Table 2 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 the execution time, the 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 should be considered.

5.4. *Neighborhood based on a 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 3.

Table 2. Results obtained using a neighborhood based on a 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	$29s$	$57s$

Table 3. Results obtained using a neighborhood based on a 2-Hamming Distance.

Problem	Fitness	# iterations	# solutions	CPU time	GPU time	Acc.
73×73	$16.4_{17.9}$	43031.7	19/50	$81s$	$10s$	$\times 8.2$
81×81	$15.5_{16.6}$	67462.5	13/50	$174s$	$16s$	$\times 11.0$
101×101	$14.2_{14.3}$	138349	12/50	$748s$	$44s$	$\times 17.0$
101×117	$13.8_{10.8}$	260130	0/50	$1947s$	$105s$	$\times 18.5$

By using this other neighborhood, in comparison with Table 2, 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 the execution time, the acceleration factor for the GPU version is really efficient (from $\times 8.2$ to $\times 18.5$). Indeed, since a large number of threads are executed, the GPU can take full advantage of the multiprocessors occupancy.

5.5. *Neighborhood based on a 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. The obtained results are collected in Table 4.

Table 4. Results obtained using a neighborhood based on a 3-Hamming Distance.

Problem	Fitness	# iterations	# solutions	CPU time	GPU time	Acc.
73×73	$2.4_{4.3}$	21360.2	35/50	$1202s$	$50s$	$\times 24.2$
81×81	$3.5_{4.4}$	43230.7	28/50	$3730s$	$146s$	$\times 25.5$
101×101	$6.2_{5.4}$	117422	18/50	$24657s$	$955s$	$\times 25.8$
101×117	$7.7_{2.7}$	255337	1/50	$88151s$	$3351s$	$\times 26.3$

In comparison with Knudsen and Meier's article [12], the results found by the generic tabu search are competitive without any use of cryptanalysis 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 the execution time,

acceleration factors using the GPU are very significant (from $\times 24.2$ to $\times 26.2$).

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

5.6. *Analysis of the Performances*

To validate the performance of our algorithms, we propose to make an analysis of the time spent by each major operation to evaluate its impact in terms of efficiency. The obtained results are reported in Fig. 3.

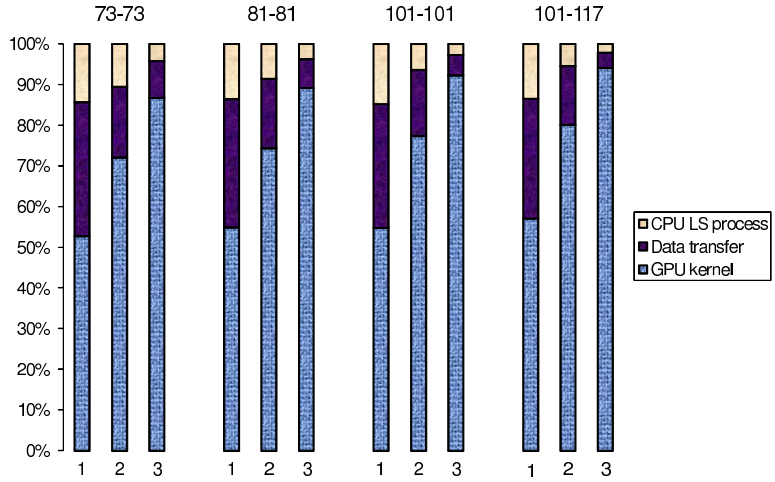

Fig. 3. Analysis of the time spent for each major operation. The 3 different neighborhoods are compared and the PPP instances are ordered according to their size.

For the first neighborhood (n neighbors), one can notice that the time spent by the data transfer is significant. For example, it represents 28% of the total execution time for the instance $m = 73, n = 73$. The same goes on for the other instances. Furthermore, regarding the time spent on the LS process on CPU, almost 15% is dedicated to this task whatever the instance size. As a consequence, since only half of the total execution time is dedicated to the GPU kernel, the amount of calculation may not be enough to fully cover the memory access latency. That is the reason why, no acceleration is provided for a tabu search based on a neighborhood with a Hamming distance of one (see Table 2).

To go on with the idea, if a neighborhood based on a Hamming distance of two is considered ($\frac{n\times(n-1)}{2}$ neighbors), one can notice that the time spent of the GPU calculation is greater than 70% for each instance. This time is almost equal to 90% for a neighborhood based on Hamming distance of three. Thereby, regarding

the time spent on both the data transfer and the LS process on CPU, it tends to decrease with 1) the number of neighbors ; 2) the instance size increase. Indeed, this can be explained by the fact that in designing LS algorithms, these two latter parameters usually have a significant influence on the total execution time and thus the amount of calculation. As a result, in accordance with the previous results (Table 3 and Table 4), algorithms based on bigger neighborhoods clearly improve the GPU occupancy i.e. the amount of calculation performed by the GPU, leading to a better global performance.

6. Discussion and Conclusion

LS algorithms based on large neighborhoods may allow to enhance the quality of provided solutions for solving real-world problems [2]. However, the exploitation of such structures is possible only by using a great computing power. GPU computing is recently revealed as a complementary way to use the huge amount of resources at disposal and fully exploit the parallelism of neighborhoods. However, the exploitation of LS parallel models is not trivial and many issues related to the GPU memory hierarchical management of this architecture have to be considered. To the best of our knowledge, no research work has been published on LS algorithms on GPU based on different neighborhoods exploration.

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 unpractical 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 cryptanalysis heuristics.

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 [14]. 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 2.3*, 2010.
[10] D. A. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *ISCA PDCS*, 2005, pp. 41–48.
[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[12] L. R. Knudsen and W. Meier, "Cryptanalysis of an identification scheme based on the permuted perceptron problem," in *EUROCRYPT*, 1999, pp. 363–374.
[13] É. D. Taillard, "Robust taboo search for the quadratic assignment problem," *Parallel Computing*, vol. 17, no. 4-5, pp. 443–455, 1991.
[14] 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.

Appendix A Two-to-one index transformation

Let us consider a 2D abstraction in which the elements of the neighborhood are disposed in a zero-based indexing 2D representation in a similar way that a lower triangular matrix. Let n be the size of the solution representation and let $m = \frac{n \times (n-1)}{2}$ be the size of its neighborhood. Let i and j be the indexes of two elements to modify in a binary encoding. A candidate neighbor is then identified by both i and j indexes in the 2D abstraction. Let $f(i, j)$ be the corresponding index in the 1D neighborhood fitnesses structure. Fig. A1 gives through an example an illustration of this abstraction.

In this example, $n = 6, m = 15$ and the neighbor identified by the coordinates ($i = 2$, $j = 3$) is mapped to the corresponding 1D array element $f(i, j) = 9$.

The neighbor represented by the (i, j) coordinates is known, and its corresponding index $f(i, j)$ on the 1D structure has to be calculated. If the 1D array size was $n * n$, the 2D abstraction would be similar to a matrix and the $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping would be:

$$f(i, j) = i \times (n - 1) + (j - 1)$$

Since the 1D array size is $m = \frac{n \times (n-1)}{2}$, in the 2D abstraction, elements above the diag-

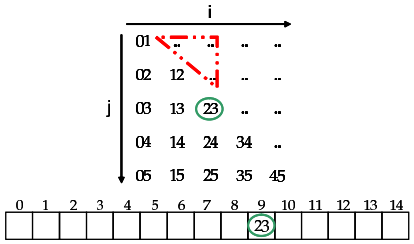

Fig. A1. $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping.

onal preceding the neighbor must not be considered (illustrated in Fig. A1 by a triangle). The corresponding mapping $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is therefore:

$$f(i,j) = i \times (n-1) + (j-1) - \frac{i \times (i+1)}{2} \tag{A.1}$$

Appendix B One-to-two index transformation

Let us consider the 2D abstraction previously presented. If the element corresponding to $f(i,j)$ in the 2D abstraction has a given i abscissa, then let k be the distance plus one between the $i+1$ and $n-2$ abscissas. If k is known, the value of i can be deduced:

$$i = n - 2 - \lfloor \frac{\sqrt{8X+1} - 1}{2} \rfloor \tag{B.1}$$

Let X be the number of elements following $f(i,j)$ in the neighborhood index-based array numbering:

$$X = m - f(i,j) - 1 \tag{B.2}$$

Since this number can be also represented in the 2D abstraction, the main idea is to maximize the distance k such as:

$$\frac{k \times (k+1)}{2} \leq X \tag{B.3}$$

Fig. B1 gives an illustration of this idea (represented by a triangle).

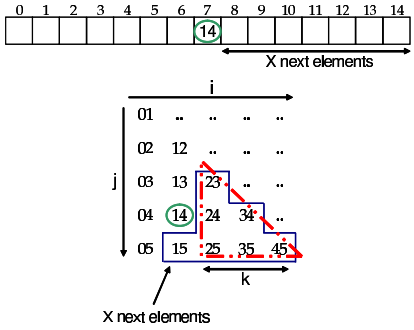

Fig. B1. $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping.

Resolving (B.3) gives the greatest distance k:

$$k = \lfloor \frac{\sqrt{8X+1} - 1}{2} \rfloor \tag{B.4}$$

A value of i can then be calculated according to (B.1). Finally, by using (A.1) j can be given by:

$$j = f(i,j) - i \times (n-1) + \frac{i \times (i+1)}{2} + 1 \tag{B.5}$$

$\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping is also done.

Appendix C One-to-three index transformation

$f(x,y,z)$ is a given index of the 1D neighborhood fitnesses structure and the objective is to find the three indexes x, y and z. Let n be the size of the solution representation and $m = \frac{n \times (n-1) \times (n-2)}{6}$ be the size of the neighborhood. The main idea is to find in which plan (coordinate z) corresponds the given element $f(x,y,z)$ in the 3D abstraction. If this corresponding plan is found, then the rest is similar as the $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping for the one-to-two index transformation previously seen. Figure C1 illustrates an example of the 3D abstraction.

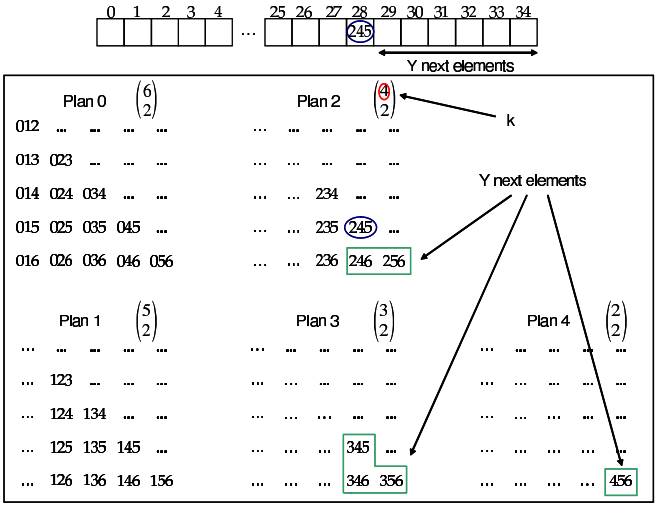

Fig. C1. $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ mapping.

In this representation, since each plan is a 2D abstraction, the number of elements in each plan is the number of combinations C_k^2 where $k \in \{2,3,\ldots,n-1\}$ according to each plan. For a specific neighbor, if a value of k is found, then the value of the corresponding plan z is:

$$z = n - k - 1 \tag{C.1}$$

For a given index $f(x,y,z)$ belonging to the plan k in the 3D abstraction, the number of elements contained in the following plans is C_k^2 (also equal to $\frac{k \times (k-1) \times (k-2)}{6}$).

Let Y be the number of elements following $f(x,y,z)$ in both the 1D neighborhood fitnesses structure and the 3D abstraction:

$$Y = m - f(x,y,z)$$

Then the main idea is to minimize k such as:

$$\frac{k \times (k-1) \times (k-2)}{6} >= Y \tag{C.2}$$

By reordering (C.2), in order to find a value of k, the next step is to solve the following equation:

$$k_1^3 - k_1 - 6Y = 0 \tag{C.3}$$

Cardano's method in theory allows to solve cubic equation. Nevertheless, in the case of finite discrete machine, this method can lose precision especially for big integers. As a consequence, a simple Newton-Raphson method for finding an approximate value of k_1 is enough for our problem. Indeed, this iterative process follows a set guideline to approximate one root, considering the function, its derivative, an initial arbitrary k_1-value and a certain precision (see Algorithm 2).

Algorithm 2 Newton-Raphson method for solving $k_1^3 - k_1 - 6Y = 0$.

1: $k_1 \leftarrow initial\_value$;
2: **repeat**
3: $\quad term \leftarrow (k_1 * k_1 * k_1 - k_1 - 6 * Y)$ / $(3 * k_1 * k_1 - 1)$;
4: $\quad k_1 \leftarrow k_1 - term$;
5: **until** $|term$ / $k_1| > precision$

Finally, since the minimization of k in (C.2) is expected, the value of k is:

$$k = \lceil k_1 \rceil$$

Then a value of z can be deduced with (C.1). At this step, the plan corresponding to the element $f(x, y, z)$ is known. The next steps for finding x and y are identically the same as the one-to-two index transformation with a change of variables.

First, the number of elements preceding $f(x, y, z)$ in the neighborhood index-based array numbering is exactly:

$$nbElementsBefore = m - \frac{(k+1) \times k \times (k-1)}{6}$$

Second, the number of elements contained in the same plan z as $f(x, y, z)$ is:

$$nbElements = \frac{k \times (k-1)}{2}$$

Finally the index of the last element of the plan z is:

$$lastElement = nbElementsBefore + nbElements - 1$$

As a result, the one-to-two index transformation is applied with a change of variables:

$$f(i, j) = f(x, y, z) - nbElementsBefore$$

$$n' = n - (z + 1)$$

$$X = lastElement - f(x, y, z)$$

After performing this transformation, a value of x and y can be deduced:

$$x = i + (z + 1)$$

$$y = j + (z + 1)$$

$\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ mapping is done.

Appendix D Three-to-one index transformation

x, y and z are known and its corresponding index $f(x, y, z)$ must be found. According to the 3D abstraction, since a value of z is known, k can be calculated:

$$k = n - 1 - z$$

Then the number of elements preceding $f(x, y, z)$ in the neighborhood index-based array numbering can be also deduced.

If each plan size was $(n-2) * (n-2)$, each 2D abstraction would be similar to a matrix and the $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping would be:

$$f_1(x, y, z) = z \times (n-2) \times (n-2) + (x-1) \times (n-2) + (y-2) \quad \text{(D.1)}$$

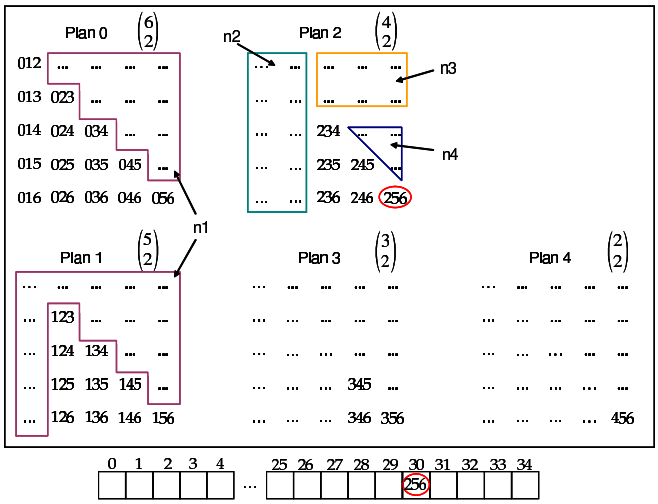

Fig. D1. $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping.

Since each 2D abstraction is some kind of triangular matrix, some elements must not be considered. The advantage of the 3D abstraction is that these elements can be found by geometric construction (see Fig. D1).

First, given a plan z, the number of elements in the previous plans to not consider is:

$$n1 = z \times (n-2) \times (n-2) - nbElementsBefore$$

Second, the number of elements on the left side to not consider in the plan z is:

$$n2 = z \times (n-2)$$

Third, the number of elements on the upper side to not consider in the plan z is:

$$n3 = (y-z) \times (n-k-1)$$

Fourth, the number of elements on the upper triangle above $f(x,y,z)$ to not consider is:

$$n4 = \frac{(y-z) \times (y-z-1)}{2}$$

Finally a value of $f(x,y,z)$ can be deduced:

$$f(x,y,z) = f_1(x,y,z) - n1 - n2 - n3 - n4 \tag{D.2}$$

$\mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping is also done.