ParadisEO-MO-GPU: a Framework for Parallel GPU-based Local Search Metaheuristics

Nouredine Melab
Inria Lille, CNRS-LIFL
Université Lille 1
Villeneuve d'Ascq - France
Nouredine.Melab@lifl.fr

Thé Van Luong
HEIG-VD
Yverdon-les-Bains
Switzerland
the-van.luong@heig-vd.ch

Karima Boufaras
Inria Lille, CNRS-LIFL
Université Lille 1
Villeneuve d'Ascq - France
boufaras.karima@gmail.com

El-Ghazali Talbi
Inria Lille, CNRS-LIFL
Université Lille 1
Villeneuve d'Ascq - France
El-Ghazali.Talbi@lifl.fr

ABSTRACT

In this paper, we propose a pioneering framework called ParadisEO-MO-GPU for the reusable design and implementation of parallel local search metaheuristics (S-Metaheuristics) on Graphics Processing Units (GPU). We revisit the ParadisEO-MO software framework to allow its utilization on GPU accelerators focusing on the parallel iteration-level model, the major parallel model for S-Metaheuristics. It consists in the parallel exploration of the neighborhood of a problem solution. The challenge is on the one hand to rethink the design and implementation of this model optimizing the data transfer between the CPU and the GPU. On the other hand, the objective is to make the GPU as transparent as possible for the user minimizing his or her involvement in its management. In this paper, we propose solutions to this challenge as an extension of the ParadisEO framework. The first release of the new GPU-based ParadisEO framework has been experimented on the permuted perceptron problem. The preliminary results are convincing, both in terms of flexibility and easiness of reuse at implementation, and in terms of efficiency at execution on GPU.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

Keywords

Local search, Metaheuristics, Parallel computing, GPU

1. INTRODUCTION

In practice, combinatorial optimization problems become more and more CPU time consuming and their modeling is continuously evolving. Therefore, there is a clear need to a software framework providing efficient parallel near-optimal optimization methods to solve large size problems. Using near-optimal search methods such as metaheuristics is a popular and efficient approach to deal with the highly combinatorial nature of industrial problems. Even if metaheuristics allow to significantly reduce the size of the search space of large problem instances, the search time remains prohibitive.

Massively parallel computing, based for instance on GPU accelerators, is required. Using a software framework is on the one hand a flexible way to deal with the evolution of modeling minimizing the design and implementation effort [3, 10]. On the other hand, it is an efficient mean to provide a transparent access to parallelism.

Metaheuristics are either single-solution namely S-Metaheuristics (e.g. local search methods) or population-based namely P-Metaheuristics (e.g. evolutionary algorithms). The focus in this paper is on S-Metaheuristics. During these two last decades, different parallel approaches and implementations have been proposed for S-Metaheuristics using Massively Parallel Processors [4], Networks or Clusters of Workstations [5] and Shared Memory or SMP machines [6]. These contributions have been later revisited for large-scale computational grids [15].

Recently, GPU accelerators have emerged as a new powerful support for massively parallel computing. A pioneering work on GPU-based S-Metaheuristics [8] has been proposed. Such experience has shown that parallel combinatorial optimization on GPU is not straightforward and requires a huge effort at design as well as at implementation level. The design of GPU-aware S-Metaheuristics often involves the cost of a sometimes painful apprenticeship of parallelization techniques and GPU computing technologies. In order to free from such burden those who are unfamiliar with those advanced features, optimization frameworks must integrate the up-to-date parallelization techniques and allow their transparent exploitation and accelerators. There exist some frameworks for the reusable design and implementation of S-Metaheuristics. However, these frameworks are often dedicated to a specific method [2] and are rarely parallel.

To the best of our knowledge, there does not exist any software framework for GPU-based S-Metaheuristics. In [3], the authors have proposed a framework called ParadisEO dedicated to the reusable design of parallel and distributed metaheuristics for only dedicated parallel hardware platforms. Later, they have extended the framework in [10] to dynamic and heterogeneous large-scale environments using Condor-MW middleware and in [14] to computational grids using Globus.

In this paper, we extend ParadisEO-MO[1] (ParadisEO for S-Metaheuristics) to deal with GPU accelerators. The challenges and contributions consist in (1) rethinking the parallel models provided into the framework to manage efficiently the hierarchical organization of the memories (different latencies and sizes) of the GPU device as well as the interaction of this latter with the CPU ; (2) making the GPU as transparent as possible for the user minimizing his or her involvement in its management. In this paper, we propose solutions to these challenges as an extension of the ParadisEO framework.

The focus is on the iteration-level parallel model of S-Metaheuristics which consists in exploring in parallel the neighborhood of a problem solution. The first release of the new GPU-based ParadisEO framework has been implemented using C++ and CUDA [11] and then experimented on the permuted perceptron problem (PPP). The preliminary results are convincing, both in terms of flexibility and easiness of reuse at implementation, and in terms of efficiency at execution on GPU.

The remainder of the paper is organized as follows. Section 2 highlights the principles of parallel iteration-level S-Metaheuristics and their challenges when using GPU computing. In Section 3, we first describe the major design features and architecture of ParadisEO. We then present the design and implementation of ParadisEO-MO on top of GPU called ParadisEO-MO-GPU. Section 4 shows and comments some experimental results obtained with ParadisEO-MO-GPU on the PPP. In Section 5, we conclude the paper and draw some perspectives of the presented work.

2. PARALLEL GPU-BASED S-METAHEURISTICS

In this section, we first present the principles of S-Metaheuristics and their associated parallel models focusing on the parallel exploration of the neighborhood. After that, we present the challenging issues to be dealt with for the GPU-acceleration of this latter.

2.1 Principles of S-Metaheuristics

S-Metaheuristics could be viewed as "walks through neighborhoods" meaning search trajectories through the exploration space of the problem being solved. Starting from an initial solution, the "walks" are performed by an iterative procedure (see Algorithm 1) that improves the current solution until a stopping criterion is met.

[1]ParadisEO-MO (http://paradiseo.gforge.inria.fr) is an open source framework originally intended to the design and deployment of parallel hybrid local search metaheuristics on dedicated clusters and networks of workstations, shared-memory machines and computational grids.

Algorithm 1 Local search pseudo-code

```
1: Generate($s_0$);
2: *Specific LS pre-treatment*
3: $t := 0$;
4: **repeat**
5:     $m(t)$ := SelectMove($s(t)$);
6:     $s_{t+1}$ := ApplyMove($m(t), s(t)$);
7:     *Specific LS post-treatment*
8:     $t := t + 1$;
9: **until** Termination_criterion($s(t)$)
```

At each iteration of the procedure, the neighborhood of the current solution is generated and evaluated. The evaluation of solutions is performed using a cost function. This latter associates a fitness value to each solution indicating its suitability to the problem. Based on the evaluation of the neighborhood, the best solution is selected to become the current solution. The process is repeated until a stopping criterion is found. This criterion could be a fixed number of iterations or a convergence criterion.

The design of an efficient and effective S-Metaheuristic requires the definition of at least a solution representation or encoding scheme, a cost function, a neighborhood model and the iterative procedure. According to the problem to be solved, three major solution encodings may be used: binary encoding (e.g. Knapsack, SAT), vector of discrete values (e.g. location problem, assignment problem) and permutation (e.g. TSP, scheduling problems). The cost function allows to evaluate the quality of the visited solutions. The neighborhood model determines the regions of the search space to be visited during the exploration. The iterative procedures (see Algorithm 1) have a common generic part but they have some specific (pre and post-treatment) features.

For instance, for the Tabu Search S-Metaheuristic (implemented and experimented in this work) [13] a specific list called *tabu list* is used. In Tabu Search, the best solution in the neighborhood is selected as the new current solution even if it is not improving the current solution. This policy may generate cycles, i.e. previous visited solutions could be selected again. To avoid these cycles, the algorithm maintains a short-term memory (tabu list) of the moves (solutions) recently applied (visited). In Algorithm 1, the specific pre-treatment consists in initializing the tabu list and post-treatment is an update of the tabu list with the most recently visited solutions.

2.2 Parallelism of S-Metaheuristics

Solving efficiently large size problems using S-Metaheuristics requires the use of parallel computing. Three major parallel models can be used separately or together in a hierarchical way: *algorithmic-level model*, *iteration-level model* or *solution-level model.* The first model consists in deploying and executing in parallel several S-Metaheuristics in a (non-)cooperative way with the objective to improve the robustness and effectiveness of the exploration. The iteration-level model consists in generating and/or evaluating the neighborhood in parallel at each iteration of the algorithm. The third model is based on the parallel evaluation of the cost function. Unlike the two first models, the solution-level model is problem-dependent. It is exploited when the cost function is CPU-time intensive and can be parallelized.

The iteration-level model, addressed in this paper, is a low-level Master-Worker model that does not alter the behavior of the heuristic. At the beginning of each iteration, the master generates the neighborhood of the current solution. Each worker receives from the master a set of neighbors (partition of the neighborhood). These neighbors are evaluated and returned back to the master. The neighbors can also be generated by the workers. In this case, each worker receives a copy of the current solution, generates one or several neighbor(s) to be evaluated and returned back to the master. A challenge of this model is to determine the granularity of each partition of neighbors to be allocated to each worker according to the communication delays of the network. In terms of genericity, as the model is problem-independent, it is generic and reusable.

2.3 GPU-accelerated S-Metaheuristics

For large-scale combinatorial optimization problems, the neighborhood of a solution is often extremely large. Therefore, massively parallel computing is required to generate and evaluate it. The parallel generation and evaluation of the neighborhood is a master-worker and problem independent regular data-parallel application. GPU computing is very well-suited for this kind of parallel application. In the GPU (e.g. CUDA-based) model, the master is the CPU and the workers are threads executed by the processing cores of the GPU. Using GPU computing is not straightforward especially for non-experts in parallel computing. Indeed, a GPU accelerator provides a hierarchy of memories with different sizes and access latencies. Exploiting efficiently such a hierarchy is particularly challenging in terms of data management and thread control.

The challenge is to re-think the design of the parallel exploration and evaluation of the neighborhood taking into account the GPU characteristics. Different issues have to be dealt with: (1) defining an efficient cooperation between CPU and GPU, which requires to share the work and to optimize the data transfer between the two devices; (2) GPU computing is based on hyper-threading (massively parallel multi-threading) and the order in which the threads are executed is not known. Therefore, an efficient mapping has to be defined between each neighboring candidate solution and a thread designated by a unique identifier assigned by the GPU runtime; (3) the neighborhood has to be placed efficiently on the different memories taking into account their sizes and access latencies.

In [8], the authors have proposed the design and the implementation of the parallel evaluation of the neighborhood model on GPU. The implementation of the proposed approaches has been performed outside our ParadisEO framework. From an implementation point of view, the challenge is to provide solutions to these issues in ParadisEO in a way as transparent as possible for the user.

The parallel iteration-level model is designed according to the data-parallel single program multiple data model of CUDA. In this model, a function code called kernel is sent to the GPU to be executed by a large number of threads grouped into blocks. As illustrated in Figure 1, the CPU-GPU task partitioning is such that the CPU hosts and executes the whole serial part of the local search method. The GPU is in charge of the evaluation of the neighborhood of

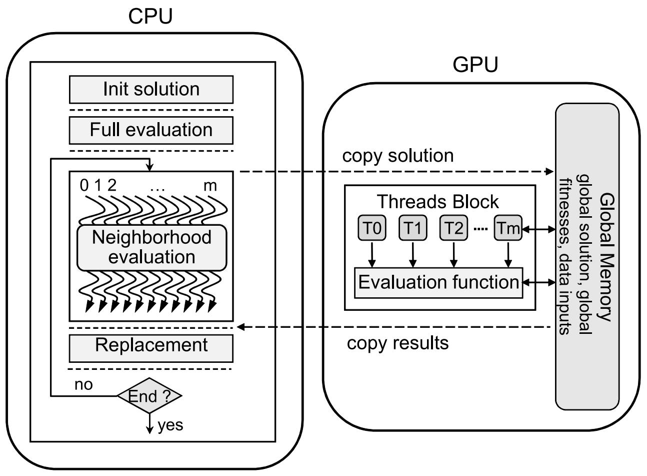

Figure 1: Parallel evaluation of the neighborhood on GPU (iteration-level). In this scheme, one thread is associated with one neighboring solution.

the current solution at each iteration. In order to minimize the cost of the data transfer from the CPU to GPU, the neighboring solutions are generated on GPU rather than on CPU. Indeed, only the current solution is sent to the GPU and each thread executes the same kernel. This is highly efficient for large neighborhoods.

The kernel consists in generating and evaluating a neighbor. A mapping function is required to allow each thread to find its corresponding (set of) neighboring solution(s). In [8], the mapping function is user-defined. In this work, we propose inside ParadisEO-MO an automatic mapping mechanism which can be exploited in a transparent way by the user application. Once all the neighboring solutions are generated and evaluated on GPU they are sent back to the CPU where the best solution is selected. The process is iterated until a stopping criterion is satisfied.

3. GPU-ENABLED PARADISEO

3.1 The ParadisEO-MO framework

ParadisEO-MO is part of ParadisEO dedicated to S-Metaheuristics such as Hill Climbing, Simulated Annealing, Tabu Search, Iterated Local Search, etc. ParadisEO [3] is a framework dedicated to the reusable design of parallel hybrid metaheuristics by providing a broad range of features including evolutionary algorithms (ParadisEO-EO), local search heuristics (ParadisEO-MO), parallel and distributed models (ParadisEO-PEO), different hybridization mechanisms, etc.

ParadisEO is a C++ LGPL extensible open source framework based on a clear conceptual separation of the metaheuristics from the problems they are intended to solve. ParadisEO is one of the rare frameworks that provide the most common parallel and distributed models. These models are portable on distributed-memory machines and shared-memory multi-processors as they are implemented using standard libraries such as MPI and Pthreads. The models can be exploited in a transparent way. One has just to instantiate their associated ParadisEO components.

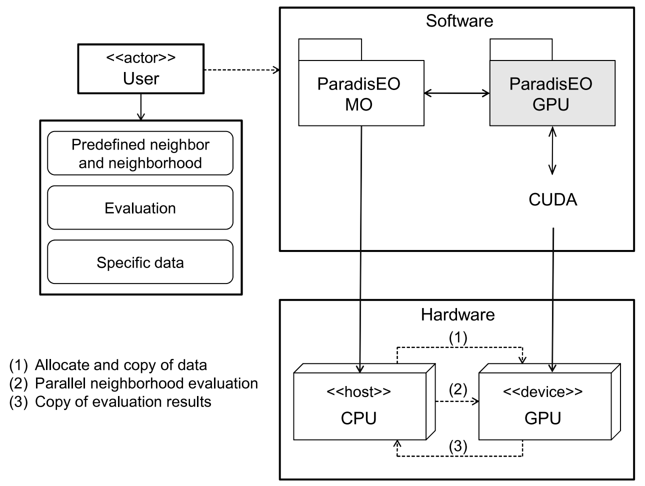
Software
<<actor>>
User
Predefined neighbor and neighborhood
Evaluation
Specific data
ParadisEO MO
ParadisEO GPU
CUDA
Hardware
(1)
(2)
(3)
<<host>>
CPU
<<device>>
GPU
(1) Allocate and copy of data
(2) Parallel neighborhood evaluation
(3) Copy of evaluation results

Figure 2: A layered architecture of ParadisEO-MO-GPU.

3.2 Architecture of ParadisEO-MO-GPU

ParadisEO-MO-GPU is a framework which is a coupling between ParadisEO-MO and CUDA for the design and implementation of reusable S-Metaheuristics on GPU. It is composed by a set of new C++ abstract and predefined classes that enables an easy and transparent development of S-Metaheuristics on GPU accelerators. The architecture of ParadisEO-MO-GPU is layered as illustrated in Figure 2.

The user layer indicates the different problem-dependent components that must be defined: input data, the evaluation function, neighbor and neighborhood representations. The software layer supplies the ParadisEO-MO components including optimization solvers embedding S-Metaheuristics. The ParadisEO-GPU module provides a CUDA interface allowing the transparent interaction with the hardware layer. The hardware layer supplies the different transparent tools provided by ParadisEO-GPU such as the allocation and copy of data or the parallel generation and evaluation of the considered neighborhood. In addition, the platform offers predefined neighborhood and mapping wrappers adapted to hardware characteristics to deal with binary and permutation problems.

The layered architecture of ParadisEO-MO-GPU has been designed in such a way that the user does not need to build his or her own CUDA code for the specific problem to be solved. Indeed, ParadisEO-GPU provides facilities for automatic execution of S-Metaheuristics on GPU. The only thing that must be user-managed is the different components described in the user level quoted above.

3.3 ParadisEO-MO-GPU Components

Figure 3 illustrates the major components of the platform. The advantage of the decomposition into components is to separate the components that must be defined by the user and those which are generic and provided in ParadisEO-MO-GPU.

Initially, to implement a sequential S-Metaheuristic, the user must overload required classes of ParadisEO-MO. The classes coding the problem-specific part are abstract classes to be specialized and implemented by the user. To GPU-enable their S-Metaheuristics, users need to derivate their own

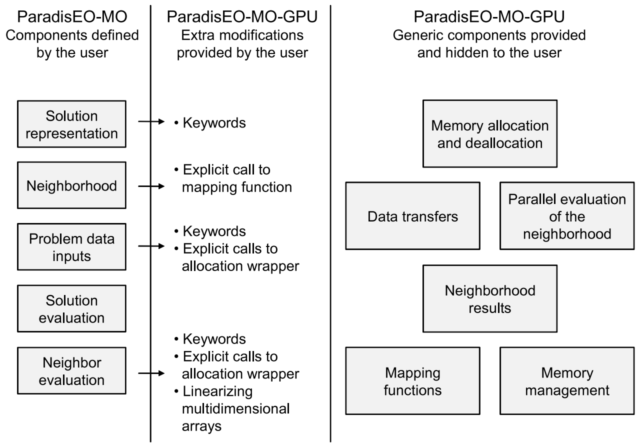
ParadisEO-MO
Components defined by the user
ParadisEO-MO-GPU
Extra modifications provided by the user
ParadisEO-MO-GPU
Generic components provided and hidden to the user
Solution representation
Neighborhood
Problem data inputs
Solution evaluation
Neighbor evaluation
• Keywords
• Explicit call to mapping function
• Keywords
• Explicit calls to allocation wrapper
• Keywords
• Explicit calls to allocation wrapper
• Linearizing multidimensional arrays
Memory allocation and deallocation
Data transfers
Parallel evaluation of the neighborhood
Neighborhood results
Mapping functions
Memory management

Figure 3: The major components of ParadisEO-MO-GPU

classes with a bench of new provided ones. In the following components, the main modifications to take into account are detailed:

- *Solution representation.* Some keywords must be specified by the user to indicate which part of the structure will be executed on GPU. Some predefined extensions for binary or permutation representations are already provided in the ParadisEO-GPU module.
- *Neighborhood.* According to the neighborhood used in the S-Metaheuristic, the user needs to make an explicit function call to a predefined mapping function. It will automatically allow to find the right association between a neighbor and a GPU thread.
- *Problem data inputs.* The user must specify which inputs of the problem to be solved will be allocated on GPU. This can be achieved by introducing additional keywords.
- *Solution evaluation.* The solution which generates the neighborhood is currently evaluated on CPU. Therefore, this component does not need to be extended.
- *Neighborhood evaluation.* In this component, structures which are likely to be allocated and managed on GPU must be indicated by the user. Every private structures specific to a neighbor (such as an array) must be declared in a static way. Furthermore, the user might translate multidimensional arrays into one-dimensional ones.

To summarize, regarding the new user-defined classes, the user must indicate the structures which are likely to be accessed on the GPU device. This strict restriction enables more efficiency and flexibility. Indeed, on the one hand, unnecessary structures for the generation and evaluation of the neighborhood should not be automatically allocated on GPU to reduce the complexity memory space. On the other hand, some additional structures required for the neighborhood evaluation might be copied for each iteration of the S-Metaheuristic whilst some others are transferred only once during the program. Such a restriction makes it possible to avoid undesirable transfers which would lead to a performance decrease.

Regarding the components supplied in the software framework, the associated classes constitute a hierarchy of classes implementing the invariant part of the code. The different features of these generic components are the following:

- *Memory allocation and deallocation.* According to the specification made by the user, this generic component enables the automatic allocation on GPU of the different structures used for the problem. Type inference and size detection are managed by this component. The same goes on for the deallocation.
- *Data transfers.* At each iteration, the transfer of the candidate solution which is used to generate the neighborhood is automatically performed from CPU to GPU. Moreover, this component also ensures the transparent copy of the neighborhood results (fitnesses) from GPU to CPU. Type inference and size detection of the different structures are also supported.
- *Parallel evaluation of the neighborhood.* This component manages the kernel of the neighborhood evaluation. Concepts involving kernel such as thread blocks and block grids are completely hidden to the user.
- *Neighborhood resulting evaluations.* This structure is manipulated in a transparent manner to store the results of the neighbors evaluated on GPU (fitnesses structure). Afterwards, this structure is sent back to the CPU to continue the local search process in a sequential manner.
- *Mapping functions.* Predefined mapping functions control the generation of the neighborhood on GPU. They consist in associating one thread with a specific neighbor. Such a mapping differs according to the used neighborhood.
- *Memory management.* Based on the user specifications, this component manages all the previous structures which are stored on global memory. This management is performed in a transparent way to the user.

The decomposition of the components in ParadisEO-MO-GPU allows to separate the features specific to S-Metaheuristics (ParadisEO-MO) from those which are related to the GPU code. This separation of concerns makes it possible to split the software framework into distinct features that overlap in functionality as little as possible.

3.4 A Case Study: Parallel Evaluation of a Neighborhood

The ParadisEO-MO-GPU execution is illustrated in Figure 4 through an UML sequence diagram. The scenario shows the design and implementation of the parallel neighborhood evaluation on GPU. At each iteration, the different stages of the parallel evaluation process on GPU are the following:

1. The neighborhood component *moCudaNeighborhood* prepares all the steps for the parallel generation of the neighborhood on GPU. The initialization consists in setting a mapping table between GPU threads and neighbors. Thereafter, the associated data are sent only once to the GPU global memory since the mapping structure does not change during the execution process of S-Metaheuristics. The last step relies on the evaluation kernel invocation. It will be informed later on its termination to retrieve the pre-computed fitnesses structure.

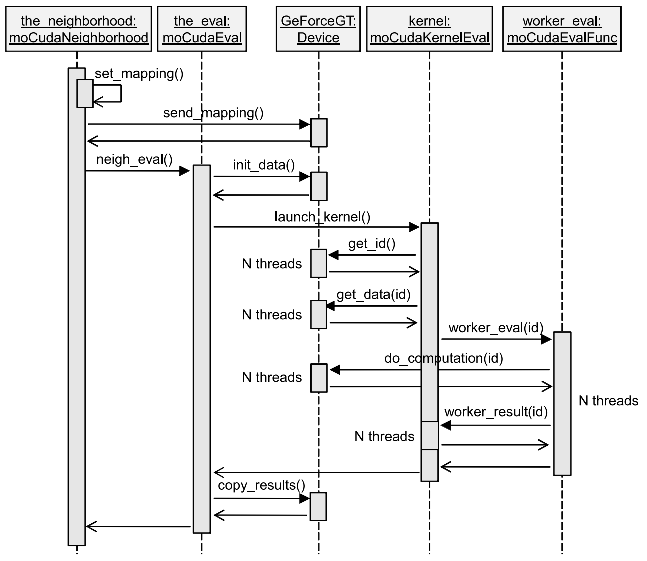

Figure 4: The parallel generation and evaluation of a neighborhood provided in ParadisEO-MO-GPU.

2. Before proceeding to the parallel evaluation, the component *moCudaEval* configures the kernel with m threads such that each thread is associated exactly with one neighbor evaluation (m designates the neighborhood size). During the first iteration, the component allocates the neighborhood fitnesses structure in which the result of the evaluated neighbors will be stored. Otherwise, in any case, it only sends to the GPU device the candidate solution which generates the neighborhood.
3. The component *moCudaKernelEval* modelizes the main body which will be executed by m concurrent threads on different input data. A first step consists in getting the thread identifier then the set of its associated data. This mechanism is done through the mapping table previously mentioned. The second step calculates the evaluation of the corresponding neighbor. Finally, the resulting fitness is stored in the corresponding index of the fitnesses structure.
4. The worker component *moCudaEvalFunc* is the specific object with computes on the GPU device the neighbor evaluation and returns back the produced result to the CPU.

Once the entire neighborhood has been carried out in parallel on GPU, the pre-calculated fitness structure is copied back to the CPU and given as input to the ParadisEO-MO module. In this way, the S-Metaheuristic continues the neighborhood exploration (iteration) on the CPU side. Instead of reevaluating each neighbor, the corresponding fitness value will be retrieved from the pre-computed fitnesses

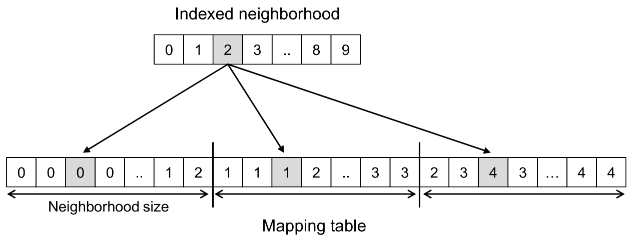

Figure 5: Automatic construction of the mapping function.

structure. Hence, this mechanism has the advantage of allowing both the deployment of any S-Metaheuristic and the use of toolboxes provided in ParadisEO-MO (e.g. statistical or fitnesses landscape analysis, checkpoint monitors, etc.)

3.5 Automatic Construction of the Mapping Function

As previously said, the advantage of generating the neighborhood on GPU is to reduce drastically the data transfers since the whole neighborhood does not have to be copied. However, the main difficulty is to find an efficient mapping between a GPU thread and neighbor candidate solution(s). In other words, the issue is to say which solution must be handled by which thread. The answer is dependent of the solution representation. In [8, 9], we provided some mappings for the main neighborhood structures of the literature. However, from an implementation point of view, they are still user-managed. Indeed, the neighborhood structure strongly depends on the target optimization problem representation.

To deal with these issues, mappings for different neighborhoods could be hard-coded in the software framework. However, such a solution does not ensure any flexibility. Hence, we propose to add a supplementary layer in terms of transparency for the deployment of S-Metaheuristics on GPU. The main idea is to find a generic mapping which is common for a set of neighborhoods. To achieve this, we provide an automatic construction of the mapping function for k-swaps and k-Hamming distance neighborhoods. Figure 5 depicts such a construction of a mapping table. In this example, each neighbor associated to a particular thread can retrieve its three corresponding indexes from the mapping table.

Considering a given vector of size n and a given neighborhood whose neighbors are composed of k indexes with k in $\{1, 2, 3, ...\}$, the size of the associated neighborhood is exactly $m = \frac{n\times(n\text{-}1)\times...\times(n\text{-}k+1)}{k!}$. The resulting mapping table associates each thread id with a set of k indexes. Each index can be respectively retrieved from the mapping table with the access pattern:

$$\{id, id + m, ..., id + (k\text{-}2) \times m, id + (k\text{-}1) \times m\}$$

The corresponding mapping table will be used at each iteration of the local search. This table is dynamically constructed on CPU according to the neighborhood size and transferred only once to the GPU global memory during the program execution.

4. APPLICATION TO THE PERMUTED PERCEPTRON PROBLEM

4.1 Problem Formulation and Resolution

In [12], Pointcheval introduced a cryptographic identification scheme based on the perceptron problem, which seems to be 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:

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 $\{\{(AV)_j/j = \{1, \ldots, m\}\}\} = S$.*

A PPP solution can be represented with a binary encoding. Part of the full evaluation of a solution can be seen as a matrix-vector product. Therefore, the evaluation of a neighbor can be performed in linear time. As the iteration-level parallel model does not change the semantics of the sequential S-Metaheuristic, the effectiveness in terms of quality of solutions is not addressed here. In [9], the authors have investigated how the increase of the size of neighborhood allows to improve the quality of the solutions. Indeed, theoretical and experimental studies have shown that the increase of the neighborhood size may improve the quality of the obtained solutions [1]. 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 reached. Thereby, in designing S-Metaheuristics, 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 massive parallelism allows to design methods based on large neighborhood structures.

In this paper, the objective is to assess the impact in terms of efficiency of an implementation done with ParadisEO-MO-GPU compared with an optimized version done outside the software framework. To measure such a difference, three neighborhoods based on increasing Hamming distances are considered for the experiments. In such neighborhoods, a neighbor is produced by flipping respectively 1, 2 and 3 bits of the candidate solution.

4.2 Experimentation with ParadisEO-MO-GPU

The considered PPP instances are the challenging ones presented in [12] for cryptanalysis applications. A Tabu Search S-Metaheuristic has been implemented in four different versions: 1) a ParadisEO-MO implementation on CPU and its counterpart on GPU; 2) an optimized CPU implementation and its associated GPU version. ParadisEO versions are pure object-based implementations whilst the optimized ones are pointer-based made outside the software framework.

Experiments have been carried out on top of an Intel Core i7 970 3.2 Ghz with a GTX 480 card (15 multiprocessors with 32 cores). To measure the acceleration factors, only a single-core CPU has been considered using the Intel i7 turbo mode (3.46 Ghz). For the two first neighborhoods, 30 executions for each different version are considered. The stopping criterion of the S-Metaheuristic has been set to 10000 iterations. The average time has been measured in

Table 1: Measures in terms of efficiency of a ParadisEO-MO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem has been considered with a neighborhood based on a Hamming distance of one (n neighbors).

Instance	ParadisEO-MO			Optimized version		
	CPU	GPU	Acc.	CPU	GPU	Acc.
73-73	0.5	1.1	×0.4	0.3	0.8	×0.4
81-81	0.6	1.2	×0.5	0.4	1.0	×0.4
101-117	1.0	1.4	×0.7	0.7	1.2	×0.6
121-137	1.4	1.5	×0.9	1.1	1.3	×0.8
151-167	2.1	1.7	×**1.2**	1.7	1.5	×**1.1**
171-187	2.7	1.9	×**1.4**	2.3	1.7	×**1.4**
201-217	3.8	2.2	×**1.7**	3.3	1.9	×**1.7**

seconds. The standard deviation is not represented since its value is pretty low. The Kolmogorov-Smirnov's test has been applied to check the normal distribution of the data set.

Table 1 reports the results obtained for the Tabu Search based on a Hamming distance of one. From the instance $m = 171$ and $n = 187$, both GPU versions start to yield positive accelerations (from ×1.1 to ×1.2). As long as the instance size increases, the acceleration factor grows accordingly (from ×1.3 to ×1.7). The acceleration factor for this implementation is not really significant. This can be explained by the fact that since the neighborhood is relatively small (n threads), the number of threads per block is not enough to fully cover the memory access latency. Furthermore, since the execution time for CPU versions is not meaningful, one can also argue on the use of GPU computing in that case. To measure the efficiency of the GPU-based implementation of this neighborhood, bigger PPP instances such as the ones used in [8] should be considered.

An experiment on larger scale concerns a Tabu Search using a neighborhood based on a Hamming distance of two. For this neighborhood, the evaluation kernel is executed by $\frac{n\times(n-1)}{2}$ threads. The obtained results from experiments are reported in Table 2.

For the first instance ($m = 73$, $n = 73$), acceleration factors are already significant (from ×8.2 and ×13.1). As long as the instance size increases, the acceleration factor grows accordingly. A peek performance is obtained for the last instance (efficient speed-ups varying from ×29.5 to ×41.8). A thorough examination of the acceleration factors points out that the performance obtained with ParadisEO-MO-GPU are not so far from an optimized implementation. The performance degradation which occurs is certainly due to the additional cost provided by ParadisEO-MO.

Indeed, regarding the two CPU versions, initially, there is already a performance gap regarding the execution time (between 68% and 86%). This difference can be explained by the overhead caused by the creation of generic objects in ParadisEO whereas the optimized version on CPU is a pure pointer-based implementation. Indeed, the Tabu Search in ParadisEO-MO is a specialized instantiation of a common template to any S-Metaheuristic whilst the optimized version is a specific Tabu Search implementation. This may also clarify the performance difference between the two different GPU counterparts in which the same phenomenon occurs. However, for such a transparent exploitation and flexibility, the obtained results are really convincing. A conclusion of this experiment indicates that the performance results of the GPU version provided by ParadisEO are not much degraded compared to the GPU pointer-based one.

As previously said, the definition of the neighborhood is a major step for the performance improvement of the algorithm. Indeed, the increase of the neighborhood size may improve the quality of the obtained solutions. However, its exploitation for solving real-world problems is possible only by using a great computing power. The next experiment illustrates that for a very large neighborhood obtained with a Hamming distance of 3. For such neighborhood, the evaluation kernel is executed by $\frac{n\times(n-1)\times(n-2)}{6}$ threads. Since the execution was too much CPU time consuming, only 5 executions have been performed. Table 3 presents the obtained results for such a large neighborhood.

In general, for the same problem instance, the obtained acceleration factors are much more important than for the previous neighborhoods. For example, for the first instance, the obtained speed-up already varies from ×17.7 to ×19.8. GPU keeps accelerated the process as long as the size grows. A very significant acceleration alternates from ×34.1 to ×53 for the biggest instance ($m = 201$, $n = 217$). Regarding the difference between ParadisEO-MO implementations and the optimized ones, the performance degradation is more important between the CPU versions (from 53% to 70%). Indeed, the increase of the neighborhood size may induce more creations of objects. This is also the same case for the performance degradation regarding the GPU counterparts. Nevertheless, according to the reported time measurements, the performance results of ParadisEO-MO-GPU are still satisfactory for such allowed transparency.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a step towards a ParadisEO framework for the reusable design and implementation of the GPU-based parallel metaheuristics. The focus is set on S-Metaheuristics and the iteration-level parallel exploration of the neighborhood of a solution. We have revisited the design and implementation of this last model in ParadisEO-MO to allow its efficient execution and its transparent use on GPU. In order to minimize the cost of the data transfer between CPU and GPU the neighborhood of the current solution is generated and evaluated on GPU at each iteration. To do that mapping functions are defined and implemented into ParadisEO-MO allowing to assign a thread identifier to each neighboring solution. These mapping functions may be used in a fully transparent way for binary and permutation-based problem representations.

The implementation in ParadisEO-MO using CUDA has been experimentally validated on a cryptographic application and compared to the same implementation realized outside ParadisEO. The experimental results show that the performance degradation that occurs between the two implementations is satisfactory. Indeed, for such a flexibility and an easiness of reuse at implementation, the results obtained with ParadisEO-MO-GPU are really promising (accelerations up to ×34). S-Metaheuristics based on large neighborhoods are unpractical on traditional machines because of their high computational cost. Hence, the use of ParadisEO-MO-GPU is a viable solution. Furthermore, we are strongly convinced that the overall performance provided by

Table 2: Measures in terms of efficiency of a ParadisEO-MO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem has been considered with a neighborhood based on a Hamming distance of two ($\frac{n\times(n-1)}{2}$ neighbors).

Instance	ParadisEO-MO			Optimized version			Perf. degradation	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
73-73	19.8	2.4	×**8.2**	17.1	1.3	×**13.1**	86%	54%
81-81	26	2.8	×**9.3**	22	1.6	×**13.8**	84%	57%
101-117	61	3.9	×**15.6**	51	2.2	×**23.2**	83%	56%
121-137	106	5.2	×**20.4**	91	2.9	×**31.3**	86%	56%
151-167	193	8.0	×**24.1**	134	4.1	×**32.7**	69%	51%
171-187	305	11.3	×**26.9**	208	5.6	×**37.1**	68%	49%
201-217	455	17.6	×**29.5**	343	8.2	×**41.8**	75%	46%

Table 3: Measures in terms of efficiency of a ParadisEO-MO-GPU implementation with an optimized version made outside the platform. The permuted perceptron problem has been considered with a neighborhood based on a Hamming distance of three ($\frac{n\times(n-1)\times(n-2)}{6}$ neighbors).

Instance	ParadisEO-MO			Optimized version			Perf. degradation	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
73-73	565	32	×**17.7**	303	15.3	×**19.8**	54%	47%
81-81	877	44	×**19.9**	464	21	×**22.1**	53%	47%
101-117	2468	99	×**24.9**	1375	50	×**27.5**	56%	50%
121-137	4887	182	×**26.8**	3137	89	×**35.2**	64%	48%
151-167	11983	401	×**29.9**	7781	190	×**41.0**	65%	49%
171-187	20239	612	×**33.1**	13089	288	×**45.4**	64%	47%
201-217	37956	1111	×**34.1**	26706	504	×**53.0**	70%	45%

ParadisEO-MO-GPU will be much better for other problems requiring more computational calculations (e.g. in [7]).

The first release of ParadisEO-MO on GPU is available on the ParadisEO website[2]. Tutorials and documentation are provided to facilitate its reuse. This release is dedicated to parallel S-Metaheuristics based on the iteration-level parallel model. In the short run, this release will be first extended to the algorithmic (multi-start) and solution-level parallel models. Second, it will be extended to other problem representations such as discrete representation and other solution methods. Third, it will be validated on a wider range of problems. In the long run, ParadisEO will be revisited following the same roadmap for population-based metaheuristics (P-Metaheuristics) such as evolutionary algorithms on GPU.

6. REFERENCES

[1] 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*, 19(3):416–428, 2007.

[2] M. Blesa, L. Hernande, and F. Xhafa. Parallel Skeletons for Tabu Search Method. In *In Proc. of ICPADS'01*, 2001.

[3] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *J. of Heuristics*, 10(3):357–380, 2004.

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

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

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

[7] T. Luong, N. Melab, and E.-G. Talbi. Parallel Hybrid Evolutionary Algorithms on GPU. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[8] T.-V. Luong, N. Melab, and E.-G. Talbi. Local Search Algorithms on Graphics Processing Units. A Case Study: the Permutation Perceptron Problem. In *EvoCOP'2010*, volume 6022 of *LNCS*, pages 264–275. Springer, 2010.

[9] T. V. Luong, N. Melab, and E.-G. Talbi. Neighborhood Structures for GPU-Based Local Search Algorithms. *Parallel Processing Letters*, 20(4):307–324, 2010.

[10] N. Melab, S. Cahon, and E.-G. Talbi. Grid computing for parallel bioinspired algorithms. *J. Parallel Distributed Computing*, 66(8):1052–1061, 2006.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[12] D. Pointcheval. A new identification scheme based on the perceptrons problem. In *EUROCRYPT*, pages 319–328, 1995.

[13] É. D. Taillard. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17(4-5):443–455, 1991.

[14] A.-A. Tantar, N. Melab, C. Demarey, and E.-G. Talbi. Building a Virtual Globus Grid in a Reconfigurable Environment - A case study: Grid5000. In *INRIA Research Report, HAL INRIA*, 2007.

[15] A.-A. Tantar, N. Melab, and E.-G. Talbi. A Comparative Study of Parallel Metaheuristics for Protein Structure Prediction on the Computational Grid. In *IEEE NIDISC/IPDPS'2007, Long Beach, California, March 26*, 2007.

[2]http://paradiseo.gforge.inria.fr