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

N. Melab, T-V. Luong, K. Boufaras and E-G. Talbi

Dolphin Project
INRIA Lille Nord Europe - LIFL/CNRS UMR 8022 - Université de Lille1
40 avenue Halley, 59650 Villeneuve d'Ascq Cedex FRANCE
[Nouredine.Melab, The-Van.Luong, Karima.Boufaras,
El-Ghazali.Talbi]@inria.fr

Abstract. This paper is a major step towards a pioneering software framework for the reusable design and implementation of parallel metaheuristics on Graphics Processing Units (GPU). The objective is to revisit the ParadisEO framework to allow its utilization on GPU accelerators. The focus is on local search metaheuristics and the parallel exploration of their neighborhood. The challenge is to make the GPU as transparent as possible for the user. The first release of the new GPU-based ParadisEO framework has been experimented on the Quadratic Assignment Problem (QAP). 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.

Keywords: Software Framework, Local Search Meta-heuristics, Parallel Computing, GPU Computing, Neighborhood Exploration.

1 Introduction

Nowadays, parallel metaheuristics have grown to be a highly useful paradigm to solve large-scale CPU time-intensive and complex combinatorial problems. Metaheuristics are either single-solution namely S-Metaheuristics (local search metaheuristics) or population-based namely P-Metaheuristics (e.g. evolutionary algorithms). The focus in this paper is on S-Metaheuristics. Recently, GPU accelerators have emerged as a new powerful support for massively parallel computing. Last year, we came up with the pioneering work on GPU-based S-Metaheuristics [1]. 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.

Indeed, 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 deployment on GPU accelerators. To the best of our knowledge, there

does not exist any software framework for GPU-based metaheuristics. In [2], we have proposed a framework called ParadisEO dedicated to the reusable design of parallel and distributed metaheuristics for only dedicated parallel hardware platforms. Later, we have extended the framework in [3] to dynamic and heterogeneous large-scale environments using Condor-MW middleware and in [4] to computational grids using Globus.

In this paper, we extend ParadisEO-MO (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 this challenge 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 [5] and then experimented on the QAP.

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 describe the major design features and architecture of ParadisEO. Section 4 presents the design and implementation of ParadisEO-MO on top of GPU called ParadisEO-MO-GPU. Section 5 shows and comments some experimental results obtained with ParadisEO-MO-GPU on the QAP. In Section 6, we conclude the paper and draw some perspectives of the presented work.

2 Parallel GPU-based S-Metaheuristics

An S-Metaheuristic is an iterative procedure which explores the neighborhood of a solution in order to improve its quality. The associated algorithm generates an initial (current) solution to the problem to be solved. This current solution is evaluated and its neighborhood is generated and evaluated. 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.

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 (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.

The challenge is to re-think the design of the parallel exploration and evaluation of the neighborhood taking into account the characteristics of GPU. 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 components; (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. From an implementation point of view, the challenge is to provide solutions to these issues in ParadisEO as transparent as possible way for the user.

3 ParadisEO-MO-based Parallel S-Metaheuristics

3.1 Parallel iteration-level model on GPU

The parallel iteration-level model is designed according to the data-parallel SPMD model of CUDA. In this model, a function code called the kernel is sent to the GPU to be executed by a large number of threads grouped into blocks. The task partitioning is such that the CPU hosts executes the whole serial part of the local search method. The GPU is in charge of the evaluation of the neighborhood of 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 defined mapping function allows each thread to find its corresponding neighboring solution. Once all the neighboring solutions are generated and evaluated they are sent back to the CPU where the best solution is selected. The process is iterated until a stopping criterion is satisfied.

3.2 The ParadisEO-MO framework

ParadisEO-MO is part of ParadisEO dedicated to S-Metaheuristics such as Hill Climbing, Simulated Annealing, Tabu Search, ILS, etc. ParadisEO [2] 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 methods (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. 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, PVM and Pthreads.

4 GPU-enabled ParadisEO

4.1 Architecture of ParadisEO-MO-GPU

ParadisEO-MO-GPU is a new framework which is a coupling between ParadisEO-MO and CUDA. It aims at deploying the S-Metaheuristics on GPU in a generic way. It is composed by a set of new C++ abstract and predefined classes that allows an easy and transparent development of S-metaheuristics on GPU accelerators. The architecture of ParadisEO-MO-GPU is layered as it is illustrated in Fig. 1.

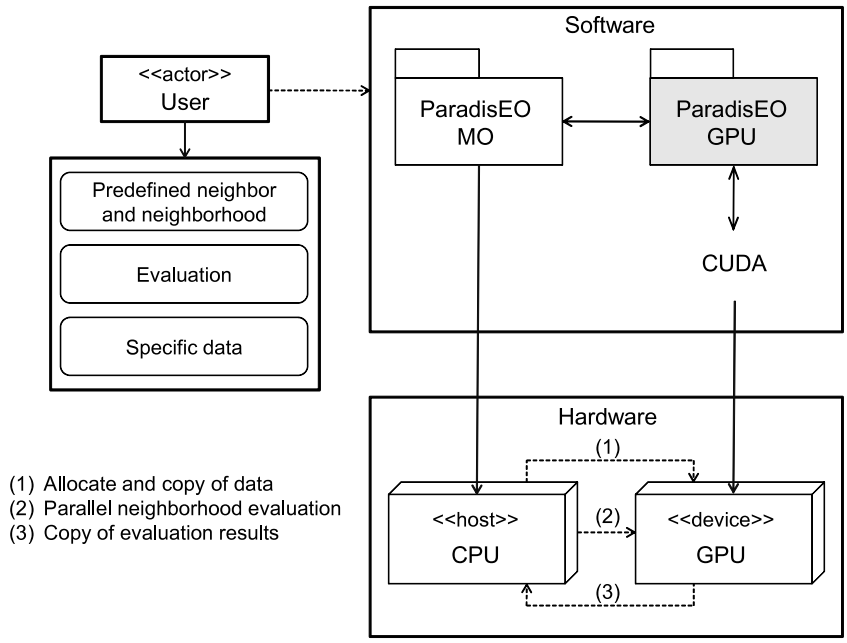

Fig. 1. A layered architecture of ParadisEO-MO-GPU.

The user level indicates the different problem-dependent components which must be defined: input data, the evaluation function, neighbor and neighborhood representations. The second level presents the ParadisEO-MO framework including optimization solvers embedding S-metaheuristics. The interaction is done with the ParadisEO-GPU module which automatically pilots the CUDA programming interface. The hardware level 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 to this, the platform proposes predefined neighborhood and mapping wrappers adapted to hardware constraints to deal with binary and permutation problems.

4.2 A case study: parallel evaluation of a neighborhood

ParadisEO-MO-GPU is illustrated in Fig. 2 through an UML sequence diagram. The scenario shows the design and implementation of parallel neighborhood evaluation of a local search on GPU. The different steps of the parallel evaluation process on GPU for each iteration are the following:

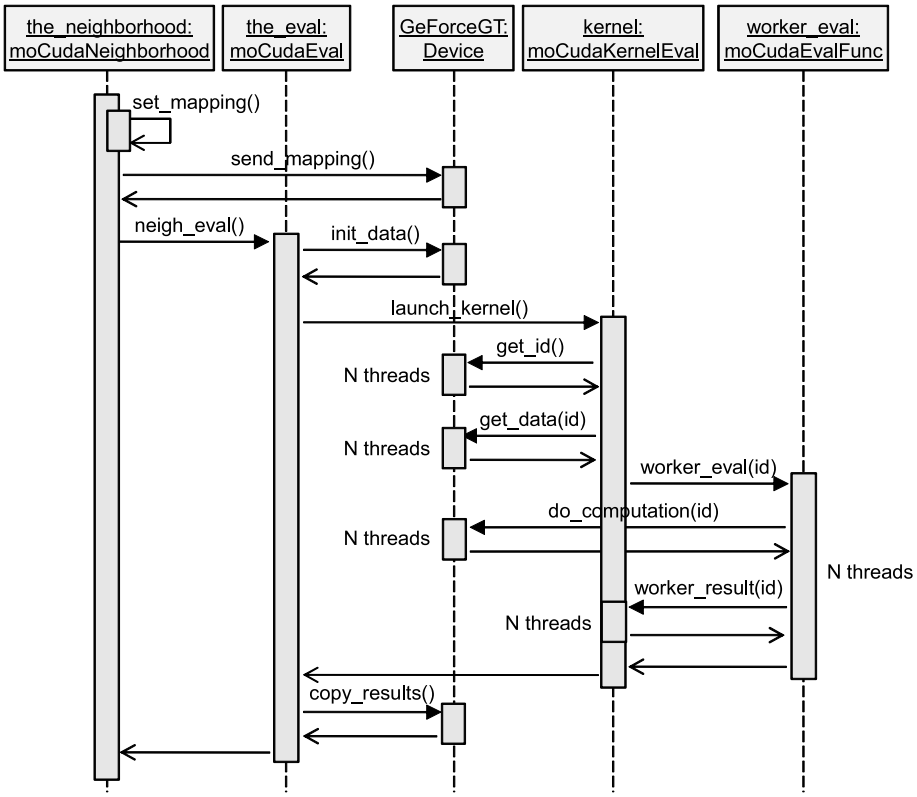

Fig. 2. The parallel generation and evaluation of a neighborhood provided in ParadisEO-MO-GPU.

1. The neighborhood *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. Then, the associated data are sent only once to the GPU since the mapping structure does not change during the execution process of local searches. The last step invokes the parallel evaluation and will be informed on its completion to retrieve the precomputed fitnesses structure.
2. Before proceeding to the parallel evaluation, the object *moCudaEval* configures a kernel with N threads such that each thread is associated exactly

with one neighbor evaluation (N designates the neighborhood size). During the first iteration, the object allocates the neighborhood fitnesses structure where 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 object *moCudaKernelEval* modelizes the main body which will be executed by N 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 thanks to the mapping table previously mentioned. The second step performs the evaluation computation of the corresponding neighbor. Finally, the resulting fitness is stored in the corresponding index of the fitnesses structure.
4. The worker *moCudaEvalFunc* is the specific object with computes on the GPU device the corresponding evaluation neighbor and returns the provided result.

Once the entire neighborhood has been performed in parallel on GPU, the precalculated fitness structure is copied back to the CPU and given as input to the ParadisEO-MO module. This way, the local search continues the neighborhood exploration (iteration) on the CPU side. Instead of evaluating again each neighbor in a sequential manner, the corresponding fitness value will only need to be retrieved from the precomputed fitnesses structure. Hence, this mechanism has the advantage of allowing both the deployment of any S-metaheuristics and the use of toolboxes provided in ParadisEO-MO (e.g. statistical or fitnesses landscape analysis, checkpoint monitors, etc.)

5 Experimentation

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

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

As the iteration-level parallel model does not change the semantics of the sequential algorithm, the effectiveness in terms of quality of solutions is not addressed here. The objective is to assess the impact in terms of efficiency of an implementation done with ParadisEO-MO-GPU compared with an optimized version made outside the platform. To achieve this, a tabu search has been implemented in 4 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 whereas the optimized ones are pointer-based.

The neighborhood is built by pair-wise exchange operations (known as the 2-exchange neighborhood) which is a standard way for permutation problems. The number of global iterations of the local search is set to 10000. The considered instances are the Taillard instances proposed in [6].

Experiments have been carried on top of an Intel Core i7 970 3.2Ghz with a GTX 480 card (15 multiprocessors with 32 cores). For measuring the acceleration factors, only a single core has been considered using the Intel i7 turbo mode (3.46Ghz). The average time has been measured in seconds for 30 runs. Results are reported in Table 1 for the different versions.

Table 1. Measures in terms of efficiency of a ParadisEO-MO-GPU implementation with an optimized version made outside the platform.

Instance	**ParadisEO-MO**			**Optimized version**			**Perf. gap**	
	CPU	GPU	Acc.	CPU	GPU	Acc.	CPU	GPU
tai30a	0.8	0.9	×0.9	0.7	0.7	×1.0	13%	23%
tai40a	1.9	1.2	×**1.6**	1.7	1.0	×**1.7**	11%	17%
tai50a	3.6	1.6	×**2.2**	3.0	1.3	×**2.3**	17%	19%
tai60a	6.0	2.0	×**3.0**	5.0	1.6	×**3.1**	17%	20%
tai80a	15.1	2.8	×**5.4**	12.3	2.1	×**5.8**	19%	25%
tai100a	31.7	3.8	×**8.3**	26.0	2.8	×**9.2**	18%	27%
tai150b	119.7	8.9	×**13.4**	98.1	6.5	×**15.1**	18%	27%

From the instance tai40a, both GPU versions start to give positive accelerations (from ×1.6 to ×1.7). The poor performance for small instances can be explained by the fact that since the neighborhood is relatively small, the number of threads per block is not enough to fully cover the memory access latency. However, as long as the instance size increases, the acceleration factor grows accordingly (e.g. from ×5.4 to ×5.8 for tai80a). Finally, efficient speed-ups are obtained for the instance tai150b. They vary between ×13.4 and ×15.1.

A thorough examination of the acceleration factors highlights that they are quite similar for the two GPU versions. The performance difference which exists is certainly due to the CPU version provided by ParadisEO-MO. Indeed, regarding the two CPU versions, initially, there is already a slight performance gap regarding the execution time. It varies between 11% and 19% according to the instance. This gap 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. This can also explain the performance difference between the two different GPU counterparts in which the same phenomenon occurs. However, for such transparent exploitation, the obtained results are really convincing. A conclusion of the experiments indicates that the performance results of the GPU version provided by ParadisEO are not so far from the GPU pointer-based one.

6 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.

The implementation in ParadisEO-MO using CUDA has been experimented on the QAP and compared to the same implementation realized outside ParadisEO. The preliminary results show that the performance gap which occurs between the two implementations is not really important. Indeed, for such flexibility and easiness of reuse at implementation, the obtained results with ParadisEO-MO-GPU are really promising (up to $\times 13$). We are strongly convinced that the overall performance provided by ParadisEO-MO-GPU will be much better for larger neighborhoods or other problems requiring more computational calculations (see [7] and [1] for previous test cases).

The first release of ParadisEO-MO on GPU is available on the ParadisEO website[1]. 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 evolutionary algorithms on GPU.

References

1. Luong, T.V., Melab, N., Talbi, E.G.: Local Search Algorithms on Graphics Processing Units. A Case Study: the Permutation Perceptron Problem. In: EvoCOP'2010. Volume 6022 of LNCS., Springer (2010) 264–275
2. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. J. of Heuristics **10**(3) (2004) 357–380
3. Melab, N., Cahon, S., Talbi, E.G.: Grid computing for parallel bioinspired algorithms. J. Parallel Distributed Computing **66**(8) (2006) 1052–1061
4. Tantar, A.A., Melab, N., Demarey, C., Talbi, E.G.: Building a Virtual Globus Grid in a Reconfigurable Environment - A case study: Grid5000 (2007)
5. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue **6**(2) (2008) 40–53
6. Taillard, É.D.: Robust tabu search for the quadratic assignment problem. Parallel Computing **17**(4-5) (1991) 443–455
7. Luong, T.V., Melab, N., Talbi, E.G.: Parallel hybrid evolutionary algorithms on gpu. In: IEEE Congress on Evolutionary Computation, IEEE (2010) 1–8

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