GPU-based Approaches for Multiobjective Local Search Algorithms. A Case Study: the Flowshop Scheduling Problem

No Author Given

No Institute Given

Abstract. Multiobjective local search algorithms are efficient methods to solve complex problems in science and industry. Even if these heuristics allow to significantly reduce the computational time of the solution search space exploration, this latter cost remains exorbitant when very large problem instances are to be solved. As a result, the use of GPU computing has been recently revealed as an efficient way to accelerate the search process. This paper presents a new methodology to design and implement efficiently GPU-based multiobjective local search algorithms. The experimental results show that the approach is promising especially for large problem instances.

1 Introduction

Real-world optimization problems are often complex and NP-hard, their modeling is continuously evolving in terms of constraints and objectives, and their resolution is time-consuming. Although near-optimal algorithms such as metaheuristics allow to reduce the temporal complexity of their resolution, they are unsatisfactory to tackle large problems.

Nowadays, GPU computing is recognized as a powerful way to achieve high-performance on long-running scientific applications [1]. Designing multiobjective local search (MLS) algorithms for solving real-world optimization problems are good challenges for GPU computing. However, only few research works related to evolutionary algorithms on GPU for monoobjective optimization exist [2–5]. Regarding existing works on LS algorithms on GPU, to the best of our knowledge, only coarse-grained implementations are provided in the context of the multi-start tabu search for monoobjective optimization [6, 7].

Indeed, a finer-grained model such that the parallel exploration of the neighborhood on GPU is not immediate and several challenges persist and are particular related to the characteristics and underlined issues of the GPU architecture and the MLS algorithms. The major issues are the efficient distribution of data processing between CPU and GPU, the thread synchronization, the optimization of data transfer between the different memories, the capacity constraints of these memories, etc.

The main objective of this paper is to deal with such issues for the re-design of parallel MLS algorithms to allow solving of large scale optimization problems on GPU architectures. In this paper, we contribute with the first results of

multiobjective local search algorithms on GPU. More exactly, we propose some new GPU-based approaches for building the parallel exploration of the neighborhood on GPU in a multiobjective context. These approaches are based on a decomposition of the GPU hierarchy allowing a clear separation between generic and problem-dependent LS features. Several challenges are dealt with: (1) the distribution of the search process among the CPU and the GPU minimizing the data transfer between them; (2) finding the efficient mapping of the hierarchical LS parallel models on the hierarchical GPU; (3) using efficiently the coalescing and texture memory in the context of MLS algorithms.

To validate the approaches presented in this paper, the flowshop scheduling problem (FSP) [8] have been considered and implemented on GPU.

The remainder of the paper is organized as follows: Section 2 highlights the principles of MLS methods and their parallel models. In Section 3, for a better understanding of the difficulties of using the GPU architecture, its characteristics are described according to a decomposition of the GPU hierarchy. Section 4 presents generic concepts for designing parallel MLS methods on GPU. In Section 5, on the one hand, efficient mappings between state-of-the-art LS structures and GPU threads model are performed. On the other hand, a depth look on the GPU memory management adapted to MLS heuristics is depicted. Section 6 reports the performance results obtained for the implemented problem mentioned above. Finally, a discussion and some conclusions of this work are drawn in Section 7.

2 Parallel Multiobjective Local Search Algorithms

2.1 Multiobjective Local Search algorithms

The existing LS methods that intend to find an approximation of the Pareto optimal set of a multiobjective optimization problem fall into two categories: scalar approaches and Pareto approaches.

Approaches belonging to the first class contains the approaches that transform a multiobjective problem into a monoobjective one or a set of such problems. Many proposed algorithms in the literature are scalar approaches. Among these methods one can find the aggregation methods, the weighted metrics, the ϵ-constraint methods ... A review of these methods is given in [9].

The second class consists in defining the acceptance of the LS according to a dominance relationship such as the Pareto dominance. The idea of Pareto approaches is to maintain an archive of non-dominated solutions, to explore the neighborhood of the solutions contained in the archive and to update the archive with the visited solutions. A complete description of the different algorithms can be found in [10].

2.2 Parallel Models of Local Search Algorithms

For non-trivial problems, executing the iterative process of a MLS on large neighborhoods requires a large amount of computational resources. In general, evaluating a fitness function for each solution is frequently the most costly operation

of the MLS. Consequently, a variety of algorithmic issues are being studied to design efficient MLS heuristics. Parallelism arises naturally when dealing with a neighborhood, since each of the solutions belonging to it is an independent unit. Parallel design and implementation of metaheuristics have been studied as well on different architectures [11, 12].

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

- *Solution-level Parallel Model.* A focus is made on the parallel evaluation of a single solution. That model is particularly interesting when the evaluation function can be itself parallelized as it is CPU time-consuming and/or IO intensive. In that case, the function can be viewed as an aggregation of a given number of partial functions.
- *Iteration-level Parallel Model.* This model is a low-level Master-Worker model that does not alter the behavior of the heuristic. Exploration and evaluation of the neighborhood are made in parallel. At the beginning of each iteration, the master duplicates the current solution between parallel nodes. Each of them manages some candidates and the results are returned back to the master.
- *Algorithmic-level Parallel Model.* Several LS algorithms are simultaneously launched for computing better and robust solutions. They may be heterogeneous or homogeneous, independent or cooperative, start from the same or different solution(s), configured with the same or different parameters.

The solution-level model is problem-dependent and does not present many generic concepts. In this paper, we focus only on the fine-grained problem-independent model: the iteration-level. Indeed, unlike the algorithmic-level, the iteration-level can be seen as an acceleration model which does not change the semantics of the algorithm.

3 GPU Computing for Metaheuristics

Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multithreaded and many-core environment. Indeed, this architecture provides tremendous computational horsepower and very high memory bandwidth compared to traditional CPUs. Since more transistors are devoted to data processing rather than data caching and flow control, GPU is specialized for compute-intensive and highly parallel computation. A complete review of GPU architectures can be found in [1].

The adaptation of MLS algorithms on GPU requires to take into account at the same time the characteristics and underlined issues of the GPU architecture and the LS parallel models. Mapping existing parallel models to the GPU in an efficient way involves a clear understanding of GPU characteristics. In this section, we propose a decomposition of the GPU adapted to the parallel iteration-level model allowing to identify the different challenges that must be dealt with.

3.1 General GPU Model

In general-purpose computing on graphics processing units, the CPU is considered as a host and the GPU is used as a device coprocessor. This way, each GPU has its own memory and processing elements that are separate from the host computer. Data must be transferred between the memory space of the host and the memory of GPU during the execution of programs.

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.

Regarding the iteration-level parallel model, since the evaluation of neighboring candidates is often the most time-consuming part of MLSs, it must be done in parallel on GPU. Therefore, according the the Master-Worker paradigm, a kernel is associated with the evaluation of the neighborhood and the CPU controls the whole sequential part of the LS process.

However, memory transfer from the CPU to the device memory is a synchronous operation which is time consuming. Indeed, bus bandwidth and latency between the CPU and the GPU can significantly decrease the performance of the search. As a result, one of the challenges is to optimize the data transfer from CPU to GPU.

3.2 Parallelism Control: GPU threads model

The kernel handling is dependent of the general-purpose language. For instance, CUDA or OpenCL are parallel computing environments which provide an application programming interface for GPU architectures [13] [14]. Indeed, these toolkits introduce a model of threads which provides an easy abstraction for SIMD architecture. The concept of a GPU thread does not have exactly the same meaning as a CPU thread. 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. Blocks can be organized into a one-dimensional or two-dimensional grid of thread blocks, and threads inside a block are grouped in a similar way. All the threads belonging to the same thread block will be assigned as a group to a single multiprocessor. Thereby, a unique *id* can be deduced for each thread to perform computation on different data.

Regarding MLS algorithms, a move which represents a particular neighbor candidate solution can also be associated with a unique *id*. However, according to the solution representation of the problem, finding a corresponding *id* for each move is not straightforward. As a consequence, another challenging issue is to find an efficient mapping between GPU threads and LS moves.

3.3 Memory Management: Kernel Management

From a hardware point of view, graphics cards consist of streaming multiprocessors, each with processing units, registers and on-chip memory. Since multiprocessors are used according to the SPMD model, threads share the same code and have access to different memory areas.

Communication between the CPU host and its device is done through the global memory. Since this memory is not cached and its access is slow, one needs to minimize accesses to global memory (read/write operations) and reuse data within the local multiprocessor memories. Graphics cards provide also read-only texture memory to accelerate operations such as 2D or 3D mapping. Constant memory is read only from kernels and is hardware optimized for the case where all threads read the same location. Shared memory is a fast memory located on the multiprocessors and shared by threads of each thread block. This memory area provides a way for threads to communicate within the same block. Registers among streaming processors are private to an individual thread, they constitute fast access memory.

The memory management on GPU is problem-specific. A clear understanding of the characteristics described above is required to provide an efficient implementation of the evaluation function. In other words, a last challenge consists in finding the association of the different LS structures with the different available memories to obtain the best performances.

4 Design of Local Search Algorithms on GPU

In this section, the focus is on the re-design of the iteration-level parallel model. Designing parallel LS model is a great challenge as nowadays there is no generic GPU-based MLS algorithms to the best of our knowledge.

4.1 A General Model for LS Algorithms

According to the SPMD model, multiple autonomous processors simultaneously execute the same program at independent points. Therefore, the mapping between the LS iteration-level parallel model and the GPU model becomes quiet natural. First, the CPU sends the number of expected running threads to the GPU, then candidate neighbors are evaluated on GPU, and finally newly evaluated solutions are returned back to the host. This model can be seen as a cooperative model where the GPU is used as a coprocessor in a synchronous manner. The resource-consuming part i.e. the generation and evaluation kernel, is calculated by the GPU and the rest is handled by the CPU.

4.2 Generation of the Neighborhood

As quoted above, CPU/GPU communication might be a major bottleneck in the performance of GPU applications. One of the major challenges is to optimize

the data transfer between the CPU and the GPU. In other words, one of the issues which remains is to say where the neighborhood must be generated. For doing that, basically there are two different approaches:

- *Generation of the neighborhood on CPU and its evaluation on GPU.* At each iteration of the LS process, the neighborhood is generated on the CPU side and its associated structure storing the solutions is copied on GPU. This approach is the most straightforward since the mapping between a thread and a neighbor is direct. It has been widely used in the context of evolutionary algorithms on GPU [2–5].
- *Generation of the neighborhood and its evaluation on GPU.* In the second approach, the neighborhood is generated on GPU. This generation is performed in a dynamic manner which implies that no explicit structure needs to be allocated. For doing that, a neighbor is considered as a slight variation of the candidate solution which generates the neighborhood. Thereby, only the representation of this candidate solution must be copied from the CPU to the GPU. The advantage of such approach is to drastically reduce the data transfers since the whole neighborhood does not have to be copied. However, as previously said, finding a mapping between a thread and a neighbor might be challenging.

Even if the first approach is the most natural one, applying it on MLS on GPU will naturally result in a lot of data transfers for large neighborhoods, leading to a great loss of performance. That is the reason why, in the rest of the paper we will consider the second approach. An analysis of the data transfers for the two different approaches might be done but it is not the scope of this paper.

4.3 The Proposed GPU-based Algorithm

Adapting traditional MLS methods to GPU is not a straightforward task because the hierarchical memory management on GPU has to be handled. We propose (see algorithm 1) a methodology to adapt MLS methods on GPU in a generic way. The given template is applicable to most of scalar and Pareto approaches.

First of all, memory allocations on GPU are made: data inputs and candidate solution of the problem must be allocated (lines 4 and 5). Since GPUs require massive computations with predictable memory accesses, a structure has to be allocated for storing the results of the evaluation of each neighbor (neighborhood fitnesses structure) at different addresses (line 6). In the case of Pareto approaches, this structure can represent different objective vectors. Additional solution structures which are problem-dependent can also be allocated to facilitate the computation of the evaluation function (line 7). Second, problem data inputs, initial candidate solution and additional structures associated with this solution have to be copied on the GPU (lines 8 to 10). It is important to notice that problem data inputs (e.g. a matrix in TSP) are a read-only structure and never change during all the execution of LS algorithms. Therefore, their associated memory are copied only once during all the execution. Third, comes the

Algorithm 1 Multiobjective Local Search Template on GPU

```
 1: Choose an initial solution
 2: Evaluate the solution
 3: Specific MLS 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 the GPU kernel do
13:       Complete or delta evaluation of the candidate solution
14:       Insert the resulting fitness into the neighborhood fitnesses structure
15:    end for
16:    Copy the neighborhood fitnesses structure on CPU host memory
17:    Specific MLS solution selection strategy on the neighborhood fitnesses structure
18:    Specific MLS 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
```

parallel iteration-level, in which each neighboring solution is generated, evaluated and copied into the neighborhood fitnesses structure (from lines 12 to 15). Fourth, since the order in which candidate neighbors are evaluated is undefined, the 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 (lines 17 and 18). For instance, the archiving of non-dominated solutions might be done in the case of Pareto approaches. Finally, after a new candidate has been selected, this latter and its additional structures are copied to the GPU (lines 19 and 20). The process is repeated until a stopping criterion is satisfied.

5 Neighborhood Generation and Memory Management

5.1 Efficient Mappings of Neighborhood Structures on GPU

Since the generation of the neighborhood is done on GPU to reduce multiple data transfers, the issue is to say which solution must be handled by which thread. The answer is dependent of the solution representation. Indeed, the neighborhood structure strongly depends on the target optimization problem representation. In the following, we provide a methodology to deal with different structures of the literature.

Binary Representation: A solution is coded as a vector of bits. The neighborhood representation for binary problems is based on the Hamming distance where a given solution is obtained by flipping one bit of the solution.

A mapping between LS neighborhood encoding and GPU threads is quiet trivial. Indeed, on the one hand, for a binary vector of size n, the size of the neighborhood is exactly n. On the other hand, threads are provided with a unique *id*. As a result, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping is straightforward.

Discrete Vector Representation: This is an extension of binary encoding using a given alphabet Σ. In this representation, each variable takes its value over the alphabet Σ. Assume that the cardinality of the alphabet Σ is k, the size of the neighborhood is $(k-1) \times n$ for a discrete vector of size n.

Let id be the identity of the thread corresponding to a given solution of the neighborhood. Compared to the initial candidate solution, $id/(k-1)$ represents the position which differs from the initial solution and $id\%(k-1)$ is the available value from the ordered alphabet Σ (both using zero-index based numbering). As a consequence, a $\mathbb{N} \rightarrow \mathbb{N}$ mapping is possible.

Permutation Representation: Building a neighborhood by pairwise exchange operations is a standard way for permutation problems. Unlike the previous representations, a mapping between a neighbor and a GPU thread is not straightforward. Indeed, on the one hand, a neighbor is composed by two element indexes. On the other hand, threads are identified by a unique *id*. As a result, a $\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ mapping has to be considered to transform one index into two ones. In a similar way, a $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ mapping is required to transform two indexes into one.

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

Given i and j the indexes of two elements to be exchanged in the permutation 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 permutation size.

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

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

The proofs of these two index transformations can be found in [15]. Notice that for different neighborhood operators such as 2-opt or the insertion operator, a slight variation of these mappings is easily applicable.

5.2 Memory Management of Local Search Algorithms on GPU

In this section, the focus is on the memory management. Understanding the GPU memory organization and issues is useful to provide an efficient implementation of parallel MLS heuristics.

Memory Coalescing Issues: In GPGPU paradigm, each block of threads is split into SIMD groups of threads called *warps*. At any clock cycle, each processor of the multiprocessor selects a half-warp (16 threads) that is ready to execute the same instruction on different data. Global memory is conceptually organized into a sequence of 128-byte segments. The number of memory transactions performed for a half-warp will be the number of segments having the same addresses than those used by that half-warp.

For more efficiency, global memory accesses must be coalesced, which means that a memory request performed by consecutive threads in a half-warp is associated with precisely one segment. The requirement is that threads of the same warp must read global memory in an ordered pattern. If per-thread memory accesses for a single half-warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. Otherwise, accessing scattered locations results in memory divergence and requires the processor to perform one memory transaction per thread. The performance penalty for non-coalesced memory accesses varies according to the size of the data structure.

Regarding LS evaluation kernels, coalescing is problem-dependent and difficult when global memory accesses have a data-dependent unstructured pattern (especially for permutation representation). As a result, non-coalesced memory accesses imply many memory transactions and it can lead to a significant performance decrease for LS methods.

Texture Memory: 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. Texture memory provides a surprising aggregation of capabilities including the ability to cache global memory. Therefore, texture memory can be seen as a relaxed mechanism for the thread processors to access global memory because the coalescing requirements do not apply to texture memory accesses.

The use of texture memory is well adapted for LS algorithms since cached texture data is laid out to give 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 texture memory. The use of textures in place of global memory accesses is a completely mechanical transformation. Details of texture coordinate clamping and filtering is given in [13].

Kernel management: Table 1 summarizes the kernel memory management in accordance with the different LS structures. The inputs of the problem (e.g. 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 declared as global memory. Indeed, since only one writing operation per thread is performed at each iteration, this structure is

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	fitnesses structure
Registers	additional variables
Local memory	additional structures

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 is local to each threads block, its use is not addressed in the generic parallel iteration-level model. However, according to the specific problem, the shared memory might be used in the solution-level model where the evaluation function is divided into partial functions.

6 Application to the Flowshop Scheduling Problem

6.1 Configuration

To validate the approaches presented in this paper, the FSP have been implemented on GPU. This problem is one of the most well-known scheduling problems. The problem can be presented as a set of n jobs J_1, J_2, ..., Jn to be scheduled on m machines. Each job J_i is composed of m consecutive tasks t_{i1}, ..., t_{im}, where t_{ij} represents the jth task of the job J_i requiring the machine M_j . To each task t_{ij} is associated a processing time p_{ij} , and to each job J_i a release time r_i and a due date d_i (deadline of the job) are given.

For the following experiments, three objectives are used in scheduling tasks on different machines:

- Makespan (total completion time); $max\{C_i|i \in [1...n]\}$
- Total tardiness; $\sum_{i=1}^{n} max(0, C_i - d_i)$
- Number of jobs delayed with regard to their due date d_i

where s_{ij} represents the time at which the task t_{ij} is scheduled and $C_i = s_{im} + p_{im}$ represents the completion time of job J_i.

The problem has been implemented using a permutation representation and the neighborhood is based on a standard insertion operator. The incremental evaluation function has a time complexity of $O(n)$ and a space complexity of $O(m \times n)$. The considered instances are the Taillard instances extended by Liefooghe in a multiobjective context [16].

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. Only average execution times and acceleration factors are reported

in comparison with a single-core CPU. The objective is to evaluate the impact of a GPU-based implementation in terms of efficiency.

Experiments have been implemented on top of two different configurations. The GPU desktop cards have a different number of cores (cadenced between 600 and 700 Mhz) which determines the number of active threads being executed. The number of global iterations of the each LS is set to 10000 which corresponds to a realistic scenario in accordance with the algorithm convergence. For each algorithm, a single-core CPU implementation, a CPU-GPU, and a CPU-GPU version using texture memory (GPU_{tex}) are considered for each configuration. To build the CPU test code, the g++ compiler has been used with the -O2 optimization flag and SSE instructions. The average time has been measured in seconds for 50 runs. For an ease of reading, the standard deviation is not represented in the following tables. Regarding the results, there is no difference of the quality of the solutions provided by both CPU and GPU.

6.2 Aggregated Tabu search

For the first experiment, a tabu search based on an aggregation (or weighted) method is used for the generation of Pareto solutions. Thereby, the FSP is transformed into a monoobjective problem by combining the various objective functions into a single one in a linear way. The results are shown in Table 2. The associated standard deviation values are close to zero.

Table 2. Time measurements for the tabu search.

Instance	**Xeon 3Ghz GTX 285 240 cores**			**Core i7 3.2Ghz GTX 480 480 cores**		
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}
20-10	1.1	$3.9_{\times 0.3}$	$3.7_{\times 0.4}$	0.9	$1.9_{\times 0.5}$	$1.7_{\times 0.6}$
20-20	2.3	$7.1_{\times 0.3}$	$6.6_{\times 0.4}$	1.9	$3.3_{\times 0.6}$	$6_{\times 0.7}$
50-10	19.8	$9.4_{\times \mathbf{2.1}}$	$8.9_{\times \mathbf{2.2}}$	16.5	$5.0_{\times \mathbf{3.3}}$	$4.3_{\times \mathbf{3.8}}$
50-20	38.0	$17.4_{\times \mathbf{2.2}}$	$16.3_{\times \mathbf{2.3}}$	31.9	$9.1_{\times \mathbf{3.5}}$	$7.8_{\times \mathbf{4.1}}$
100-10	170.8	$23.6_{\times \mathbf{7.2}}$	$20.9_{\times \mathbf{8.2}}$	144.5	$12.6_{\times \mathbf{11.5}}$	$11.5_{\times \mathbf{12.6}}$
100-20	321.1	$44.1_{\times \mathbf{7.3}}$	$38.1_{\times \mathbf{8.4}}$	270.7	$23.3_{\times \mathbf{11.6}}$	$20.9_{\times \mathbf{12.9}}$
200-10	1417.4	$159.4_{\times \mathbf{8.9}}$	$144.3_{\times \mathbf{9.8}}$	1189.7	$81.88_{\times \mathbf{14.5}}$	$77.2_{\times \mathbf{15.4}}$
200-20	2644.1	$284.4_{\times \mathbf{9.3}}$	$263.9_{\times \mathbf{10.0}}$	2220.7	$147.8_{\times \mathbf{15.0}}$	$139.0_{\times \mathbf{16.0}}$

From the instance 50-10, GPU versions start to give positive accelerations for both configurations (from $\times 2.1$ to $\times 3.8$). Indeed, the slow speed-ups 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. As long as the instance size increases, the acceleration factor grows accordingly. For example, from a larger instance such as 100-10, the provided speed-ups are much better (from $\times 7.2$ to $\times 12.6$).

For each instance, in a general manner, the use of texture memory allows to provide additional acceleration. However, constraints of memory alignment in latest G200 and G400 series are relaxed in comparison with the previous cards (e.g. G80 and G90 series). As a consequence, programs running on the used cards get a better global memory performance and the benefits of using the texture memory are less evident.

Finally, efficient speed-ups are obtained for the instance 200-20. They vary between $\times 9.3$ and $\times 16$. As a consequence, parallelization on top of GPU provides an efficient way for handling large neighborhoods.

6.3 Pareto Local Search algorithms

For the next experiments, the PLS-1 (Pareto local search proposed by Paquete *et al* [10]) has been considered. At each iteration, PLS-1 selects a non-dominated solution from the unbounded archive and explores its neighborhood in an exhaustive way. The termination condition is done when all the solutions contained in the archive have been visited. For the experiments, a restart mechanism is performed while the number of iterations has not reached 10000. The solutions archiving on CPU is done in a general way where the solution to insert is compared with each of those contained in the archive.

A first algorithm $PLS_{\succ}$ have been considered where only the dominating neighbors are added to the archive. The results of the experiments are provided in Table 3. Kolmogorov-Smirnov statistical tests can be made to check the normal distribution of the different results.

Table 3. Time measurements for $PLS_{\succ}$.

Instance	**Xeon 3Ghz GTX 285 240 cores**			**Core i7 3.2Ghz GTX 480 480 cores**			**# $\not\prec$ solutions**
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	
20-10	1.1	$5.3_{\times 0.2}$	$3.7_{\times 0.3}$	1.0	$1.9_{\times 0.5}$	$1.7_{\times 0.6}$	11
20-20	2.5	$8.4_{\times 0.3}$	$6.6_{\times 0.4}$	1.9	$3.4_{\times 0.6}$	$3.0_{\times 0.6}$	13
50-10	19.6	$10.9_{\times \mathbf{1.8}}$	$8.9_{\times \mathbf{2.2}}$	16.6	$4.9_{\times \mathbf{3.4}}$	$4.3_{\times \mathbf{3.8}}$	19
50-20	39.7	$18.9_{\times \mathbf{2.1}}$	$16.5_{\times \mathbf{2.4}}$	32.7	$9.1_{\times \mathbf{3.6}}$	$7.8_{\times \mathbf{4.2}}$	20
100-10	167.0	$25.3_{\times \mathbf{6.6}}$	$21.2_{\times \mathbf{7.9}}$	139.5	$12.8_{\times \mathbf{10.9}}$	$11.7_{\times \mathbf{12.0}}$	31
100-20	329.8	$45.8_{\times \mathbf{7.2}}$	$40.4_{\times \mathbf{8.1}}$	274.9	$23.5_{\times \mathbf{11.7}}$	$21.1_{\times \mathbf{13.1}}$	34
200-10	1391.5	$161.8_{\times \mathbf{8.6}}$	$145.1_{\times \mathbf{9.6}}$	1171.5	$82.5_{\times \mathbf{14.2}}$	$77.9_{\times \mathbf{15.1}}$	61
200-20	2707.8	$307.7_{\times \mathbf{8.8}}$	$285.9_{\times \mathbf{9.4}}$	2196.3	$148.4_{\times \mathbf{14.8}}$	$139.7_{\times \mathbf{15.7}}$	71

In comparison with Table 2, similar observations can be made regarding the performance results where the maximal speed-up reaches the value $\times 15.7$ for the biggest instance. A look at the average number of non-dominated solutions obtained by the algorithm shows that this number is rather low whatever the

instance size. Therefore, this may explained why the performance results of the Pareto algorithm are similar to the aggregated tabu search. To emphasize this point, a second algorithm $PLS_{\not\prec}$ has been considered. In this version, all the non-dominated neighbors are added to the archive. Table 4 reports the different measurements.

Table 4. Time measurements for $PLS_{\not\prec}$.

Instance	**Xeon 3Ghz GTX 285 240 cores**			**Core i7 3.2Ghz GTX 480 480 cores**			**# $\not\prec$ solutions**
	CPU	GPU	GPU_{Tex}	CPU	GPU	GPU_{Tex}	
20-10	1.1	$5.4_{\times 0.2}$	$3.9_{\times 0.3}$	1.2	$2.0_{\times 0.6}$	$1.8_{\times 0.7}$	77
20-20	2.6	$8.5_{\times 0.3}$	$6.7_{\times 0.4}$	2.0	$3.4_{\times 0.6}$	$3.0_{\times 0.7}$	83
50-10	25.7	$15.1_{\times \mathbf{1.7}}$	$13.1_{\times \mathbf{1.9}}$	20.5	$8.9_{\times \mathbf{2.3}}$	$8.2_{\times \mathbf{2.5}}$	396
50-20	44.5	$24.7_{\times \mathbf{1.8}}$	$22.3_{\times \mathbf{2.2}}$	39.5	$14.1_{\times \mathbf{2.8}}$	$13.2_{\times \mathbf{3.0}}$	596
100-10	220.1	$71.0_{\times \mathbf{3.1}}$	$66.9_{\times \mathbf{3.3}}$	253.3	$51.7_{\times \mathbf{4.9}}$	$49.2_{\times \mathbf{5.1}}$	1350
100-20	386.8	$96.7_{\times \mathbf{4.0}}$	$91.3_{\times \mathbf{4.2}}$	312.5	$62.5_{\times \mathbf{5.0}}$	$59.0_{\times \mathbf{5.3}}$	1530
200-10	1631.2	$362.5_{\times \mathbf{4.5}}$	$346.3_{\times \mathbf{4.7}}$	1361.5	$223.2_{\times \mathbf{6.1}}$	$217.7_{\times \mathbf{6.4}}$	1597
200-20	2998.8	$588.0_{\times \mathbf{5.1}}$	$566.1_{\times \mathbf{5.3}}$	2672.6	$398.9_{\times \mathbf{6.7}}$	$378.0_{\times \mathbf{7.1}}$	2061

For the instance 100-10, in comparison with the previous table, one can clearly start to see the impact of the number of non-dominated solutions in terms of performance. Indeed, the acceleration factors vary between $\times 3.1$ to $\times 5.1$. The performance results are significantly reduced in comparison with the analog instance for $PLS_{\succ}$. Finally, the speed-up still grows with the size increase until reaching the value $\times 7.1$ for the last instance.

This global performance loss can be explained by the fact that the number of non-dominated solutions is more important. Indeed, one can see that $PLS_{\not\prec}$ is more time-consuming than its counterpart $PLS_{\succ}$. Therefore, the time spent on the archiving of solutions may be significant. Furthermore, this step is performed exclusively on CPU.

Table 5 confirms this point by showing a profiling of the different steps of the different algorithms. As one can see, for $PLS_{\not\prec}$, the time dedicated to the solutions archiving on CPU is more important than its counterpart $PLS_{\succ}$. As a consequence, less time is dedicated to the parallel generation and evaluation on GPU. The conclusion of these experiments indicates that since the solutions archiving is significant in the PLS process, it may be significant to proceed this archiving on GPU.

7 Discussion and Conclusion

High-performance computing based on the use of computational GPUs is recently revealed to be a good way to get at hand such computational power.

Table 5. Profiling of the two Pareto algorithms.

Instance	$\mathbf{PLS}_{\succ}$			$\mathbf{PLS}_{\not\prec}$		
	Evaluation	Archiving	LS process	Evaluation	Archiving	LS process
20-10	99.2%	0.7%	0.1%	95.2%	4.7%	0.1%
20-20	99.4%	0.5%	0.1%	98.2%	1.7%	0.1%
50-10	99.5%	0.4%	0.1%	89.7%	10.2%	0.1%
50-20	99.7%	0.2%	0.1%	94.2%	5.7%	0.1%
100-10	99.8%	0.1%	0.1%	88.2%	11.8%	0.1%
100-20	99.85%	0.1%	0.05%	93.6%	6.35%	0.05%
200-10	99.9%	0.05%	0.05%	92.3%	7.65%	0.05%
200-20	99.9%	0.05%	0.05%	94.2%	5.75%	0.05%

However, the exploitation of 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, GPU-based parallel LS approaches have never been widely investigated.

In this paper, efficient mapping of the iteration-level parallel model on the hierarchical GPU has been proposed. In the cooperation layer, the CPU manages the whole MLS process and let the GPU be used as a coprocessor dedicated to intensive calculations. Then, efficient mappings between neighborhood candidate solutions and GPU threads are necessary to generate the neighborhood on GPU and thus reduce CPU/GPU transfers. Finally, memory management is applied to the evaluation function kernel.

Apart from being generic, we proved the effectiveness of our methodology by making extensive experiments. Applying such mechanism with an efficient memory management allows to provide promising speed-ups (up to $\times 16$). We strongly believe that the overall performance could be better for other multiobjective optimization problems requiring 1) more computational calculations and 2) less resources in terms of memory (linear time complexity and quadratic space complexity for each neighbor of the FSP).

The re-design of the parallel MLS iteration-level model on GPU fits well for deterministic scalar and Pareto approaches. However, few other MLS algorithms only partially explore neighborhoods and take the first improving local neighbor that is detected. Applied to LS algorithm such as multiobjective simulated annealing, this model needs to be re-thought.

Furthermore, in the case of Pareto approaches, the experimental results have shown a global performance loss with the increase of non-dominated solutions. For other multiobjective optimization problems whose objectives are uncorrelated, this number could be huge, leading to a serious performance decrease. Even if some archiving techniques could be applied to bound the archive size, this would not completely solve the issue at all. As a consequence, for being complete, the next step of our method is to provide a SIMD parallel archiving on GPU. This way, it will allow to significantly enhance the performance of the provided algorithms. However, performing such step is challenging since it

requires to ensure additional synchronizations, non-concurrent writings and to manage some dynamic allocations on the GPU.

References

1. Ryoo, S., Rodrigues, C.I., Stone, S.S., Stratton, J.A., Ueng, S.Z., Baghsorkhi, S.S., mei W. Hwu, W.: Program optimization carving for gpu computing. J. Parallel Distribributed Computing **68**(10) (2008) 1389–1401
2. Li, J.M., Wang, X.J., He, R.S., Chi, Z.X.: An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In: Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference. (2007) 855–862
3. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: GECCO. (2007) 1566–1573
4. Wong, T.T., Wong, M.L.: Parallel evolutionary algorithms on consumer-level graphics processing unit. In: Parallel Evolutionary Computations. (2006) 133–155
5. Fok, K.L., Wong, T.T., Wong, M.L.: Evolutionary computing on consumer graphics hardware. IEEE Intelligent Systems **22**(2) (2007) 69–78
6. Zhu, W., Curry, J., Marquez, A.: Simd tabu search with graphics hardware acceleration on the quadratic assignment problem. International Journal of Production Research (2008)
7. Janiak, A., Janiak, W.A., Lichtenstein, M.: Tabu search on gpu. J. UCS **14**(14) (2008) 2416–2426
8. Taillard, E.: Benchmarks for basic scheduling problems (1989)
9. Talbi, E.G.: Metaheuristics: From design to implementation. Wiley (2009)
10. Paquete, L.: Stochastic Local Search Algorithms for Multiobjective Combinatorial Optimization: Methods And Analysis. IOS Press (2006)
11. Alba, E., Talbi, E.G., Luque, G., Melab, N.: 4. Metaheuristics and Parallelism. Wiley Series on Parallel and Distributed Computing. In: Parallel Metaheuristics: A New Class of Algorithms. Wiley (2005) 79–104
12. Melab, N., Cahon, S., Talbi, E.G.: Grid computing for parallel bioinspired algorithms. J. Parallel Distributed Computing **66**(8) (2006) 1052–1061
13. NVIDIA: CUDA Programming Guide Version 3.2. (2010)
14. Group, K.: OpenCL 1.0 Quick Reference Card. (2010)
15. Luong, T.V., Melab, N., Talbi, E.G.: Large neighborhood for local search algorithms. In: International Parallel and Distributed Processing Symposium, IEEE Computer Society (2010)
16. Liefooghe, A., Basseur, M., Jourdan, L., Talbi, E.G.: Combinatorial optimization of stochastic multi-objective problems: An application to the flow-shop scheduling problem. In Obayashi, S., Deb, K., Poloni, C., Hiroyasu, T., Murata, T., eds.: EMO. Volume 4403 of Lecture Notes in Computer Science., Springer (2006) 457–471