Parallelization Strategies for Local Search Algorithms on Graphics Processing Units

Audrey Delévacq, Pierre Delisle, and Michaël Krajecki
CReSTIC, Université de Reims Champagne-Ardenne, Reims, France

Abstract— *The purpose of this paper is to propose effective parallelization strategies for Local Search algorithms on Graphics Processing Units (GPU). We consider the distribution of the 3-opt neighborhood structure embedded in the Iterated Local Search framework. Three resulting approaches are evaluated and compared on both speedup and solution quality on a state-of-the-art Fermi GPU architecture. Solving instances of the Travelling Salesman Problem ranging from 100 to 3038 cities, we report speedups of up to 8.51 with solution quality similar to the best known sequential implementations and of up to 45.40 with a variable increase in tour length. The proposed experimental study highlights the influence of the pivoting rule, neighborhood size and parallelization granularity on the obtained level of performance.*

Keywords: TSP, ILS, Parallel Metaheuristics, 3-opt, GPU

1. Introduction

Iterated Local Search (ILS) is a metaheuristic that successively applies a given Local Search (LS) procedure to an initial solution and incorporates mechanisms to climb out of local optima. This method finds good solutions to many optimization problems in a reasonable time which may still remain too high in practice. However, it offers an interesting parallelization potential when extended to a population-based approach where different individuals improve their solution by executing the same algorithm on several computing units [1].

At present time, the best known implementations of the ILS framework are dedicated to conventional, CPU-based sequential and parallel architectures. However, as recent years have shown the potential to use the GPU as a high performance computational resource, it becomes important to deliver GPU-based optimization methods that are efficient in both solution quality and execution speed. However, recent research has shown that this goal is often difficult to achieve.

The purpose of this paper is to propose parallelization strategies for ILS to efficiently solve the Travelling Salesman Problem (TSP) in a GPU computing environment. The 3-opt exchange procedure used within the algorithm is decomposed on processing elements along with required data structures. Pivoting rules based on first-improvement and best-improvement schemes are also evaluated. Important algorithmic, technical and programming issues that may be encountered in this context are finally highlighted.

This paper is organized as follows. First, we present k-opt LS algorithms, the ILS metaheuristic and their application to the TSP. After a fairly complete review of the literature on parallel LS and ILS, the proposed GPU parallelization strategies are explained. Finally, a comparative experimental study is performed to evaluate the performance of the resulting algorithms.

2. Iterated Local Search for the Travelling Salesman Problem

The Travelling Salesman Problem (TSP) may be defined as a complete weighted directed graph $G = (V, A, d)$ where $V = \{1, 2, ..., n\}$ is a set of vertices (cities), $A = \{(i, j) \mid (i, j) \in V \times V\}$ is the set of arcs and $d : A \rightarrow \mathbb{N}$ is a function assigning a weight or distance d_{ij} to every arc (i, j). The objective is to find a minimum weight Hamilton cycle in G, which is a tour of minimal length visiting each city exactly once.

Local Search (LS) generally aims to iteratively improve an initial solution by local transformations, replacing a current solution by a better neighbor until no more improving moves are possible. In that case, a local optimum is reached and the procedure stops. Most well-known LS algorithms for the TSP are based on k-opt exchanges which delete k arcs of a current solution and reconnect partial tours with k other arcs in all possible ways. Figure 1 describes the specific 3-opt [2] procedure. Among the other popular ones, we may cite the 2-opt [3] and Lin-Kernighan [4] algorithms.

One of the key elements of LS is the pivoting rule which dictates the choice of the neighbor solution that will replace the current one [5]. The most commonly used methods are best-improvement and first-improvement. In the first case, all neighbors of the current solution are evaluated and the one which produces the greatest improvement is selected. In the second case, the first improving move is accepted and the others are discarded. Less popular alternatives are random-improvement and least-improvement which respectively choose a neighbor randomly or with minimal improvement of the objective function. Anderson [6] defines a parameter k associated to the number of improving moves found before choosing the best one. When $k = 1$, the algorithm uses a first-improvement strategy and

the more k increases, the more thorough is the exploration of the neighborhood. He deduces that first-improvement is generally the best choice for TSP.

In order to accelerate the execution of LS algorithms for the TSP, various mechanisms are usually used to reduce the neighborhood of the current solution. First, candidates lists, which comprise the cl nearest cities for each city, may be used when reconnecting partial tours instead of the whole set of cities. Second, fixed-radius neighbor search reconnects tours only with arcs for which the sum of weights is potentially lower than the sum of weights of the arcs to be deleted. Third, *don't look bits* are associated to each city in order to drive the search away from arcs which have recently led to unimproving moves. A complete description of these methods may be found in Bentley [7] and in Johnson and McGeoch [8].

A LS procedure becomes trapped in a local optimum when no improving moves are possible in the neighborhood of the current solution. A way to partly counter this problem is to embed it in a guiding construction such as Iterated Local Search (ILS) [9]. This metaheuristic is divided into four main steps that are highlighted in Figure 2. The first one is the generation of an initial solution S, usually with constructive heuristics or randomly. The second one is the LS procedure that is applied to S to bring it to a local optimum. The third one is a perturbation move which transforms S into S' in order to get it out of that local optimum. Finally, an acceptance criterion is evaluated to choose which solution between S and S' will be used to resume the search. The three last steps are then repeated until an end criterion is reached, for example a maximum time limit or iteration count.

```
while S is not a local optimum do
    for each combination of a, b, c ∈ [0; n] do
        Delete arcs (a,a+1), (b,b+1) and (c,c+1)
        Produce neighbors of S by reconnecting partial tours
        Evaluate neigbors of S
    Replace S by the best neighbor chosen by the pivoting rule
Return best solution S
```

Fig. 1: 3-opt LS pseudo-code.

```
Generate solution S
Apply LS procedure on S
Evaluate length L of solution S
while end criterion is not reached do
    Transform S into S' by a perturbation move
    Apply LS procedure on S'
    Evaluate length L' of solution S'
    if L' < L then replace S by S' // acceptance criterion
Return best solution S
```

Fig. 2: ILS pseudo-code.

The ILS metaheuristic is considered as one of the most powerful approximate methods for the TSP [1]. In fact, the works of Stützle and Hoos [10] and Lourenço *et al.* [9] show its competitiveness in solving various TSP problems varying from 100 to 5915 cities. However, faced to large and hard optimization problems, it may need a considerable amount of computing time and memory space to be effective in the exploration of the search space. A way to accelerate this exploration is to use parallel computing.

3. Literature review on parallel LS

Verhoeven and Aarts [11] proposed a classification that distinguishes *single-walk* and *multiple-walk* parallelization approaches for LS algorithms. In the first category, one search process goes through the search space and its steps are decomposed for parallel execution. In that case, neighbors of a solution may be evaluated in parallel (*single-step*) or several exchanges may be performed on different parts of that solution (*multiple-step*). In the second category, many search processes are distributed over processing elements and designed either as *multiple independent walks* or *multiple interacting walks*.

Johnson and McGeoch [8] defined three parallelization strategies for k-opt algorithms. The first one uses *geometric partitioning* to divide the set of cities into subgroups that are sent to different processors to be improved by a constructive algorithm and a LS procedure. As this partitioning has the drawback of isolating subgroups without reconnecting subtours intelligently, the second strategy favors *tour-based partitioning* to divide tours into partial solutions that includes a part of the edges of the current solution. The third approach is a simple parallelization of neighborhood construction and exploration.

Works on parallelization of ILS for the TSP mainly follow the population-based, multiple-walk approach where many solutions are built concurrently. Hong *et al.* [12] designed a parallel ILS which executes a total of m iterations using a pool of p solutions. Martin and Otto [13] proposed an implementation in which several solutions are computed simultaneously on different processors and the best solution replaces all solutions at irregular intervals.

Few authors have tackled the problem of parallelizing LS algorithms for TSP on GPU. Luong *et al.* [14], [15] proposed a methodology for implementing large neighborhood algorithms in which the CPU is in charge of LS processes and the GPU deals with the generation and evaluation of neighbor solutions. It was experimented on TSP with Tabu Search using a swap as the local transformation. Maximal speedup of 19.9 is reported on a 5915 cities problem but solution quality is not provided. O'Neil *et al.* [16] implemented an iterative hill climbing algorithm based on 2-opt local transformations in which random restarts are associated to threads. Maximal speedup of 61.9 is reported on a 100 cities problem. Fujimoto and Tsutsui [17] integrated a 2-opt best-improvement LS into a genetic algorithm executed

on GPU. Maximal speedup of 24.20 is obtained on problems of up to 493 cities. Delévacq *et al.* [18] augmented an Ant Colony Optimization algorithm with a 3-opt local search implemented on GPU. Solving TSP problems from 198 to 2103 cities, they reported speedups of up to 8.03 with solution quality similar to the best known sequential implementation.

These works provide a frame of reference for evaluating the attainable efficiency of GPU-based LS algorithms for the TSP. However, most of them overlook important issues that make it difficult to assess their effectiveness as parallel optimization methods. First, speedups are mostly provided with nonexistent or inappropriate evaluation of solution quality. The proposed implementations are also experimented on TSPs often limited to a few hundred cities without bypassing obvious memory limits of actual GPUs. In addition, as significant implementation details like the pivoting rule and the mechanisms used to accelerate the computations are not provided, algorithms can hardly be reproduced or tested on larger problems. Finally, basic local transformations are most often implemented even though the most effective methods for solving the TSP are based on the 3-opt neighborhood structure and the Lin-Kernighan algorithm [6].

As LS algorithms are key underlying components of most high-performing metaheuristics, a natural fit is to run a guiding metaheuristic on CPU while the GPU, acting as a co-processor, takes charge of running the LS procedure. However, there is still much conceptual, technical and comparative work to achieve in order to design such hybrid parallel combinatorial optimization methods. This paper aims to partially fill this gap by proposing, evaluating and comparing various parallelization strategies for the 3-opt LS on GPU.

4. GPU parallelization strategies for ILS

We propose three parallelization strategies based on the algorithm described in Figures 1 and 2. They mainly differ in the pivoting rule used to select an improving neighbor, the distribution of solutions to processing elements and the use of GPU shared memory. For the sake of completeness, we first provide a brief description of the GPU architecture and computational model.

The conventionnal NVIDIA GPU [19] architecture includes many *Streaming Multiprocessors* (SM), each one of them being composed of *Streaming Processors* (SP). On this special hardware, the *global* memory is a specific region of the *device* memory that can be accessed in read and write modes by all SPs of the GPU. It is relatively large in size but slow in access time. Each SM employs an architecture model called *SIMT* (*Single Instruction, Multiple Thread*) which allows the execution of many coordinated threads in a data-parallel fashion. *Constant* and *texture memory caches* provide faster access to device memory but are read-only. The constant memory is very limited in size whereas texture memory size can be adjusted in order to occupy the available device memory. All SPs can read and write in their *shared memory*, which is fast in access time but small in size and local to a SM. It is divided into memory banks of 32-bits words that can be accessed simultaneously. *Registers* are the fastest memories available on the GPU but involve the use of slow local memory when too many are used.

In the CUDA programming model [19], the GPU works as a co-processor of a conventional CPU. It is based on the concept of kernels, which are functions (written in C) executed in parallel by a given number of CUDA threads. These threads are grouped together into *blocks* that are distributed on the GPU SMs to be executed independently of each other. However, the number of blocks that a SM can process at the same time (*active blocks*) is restricted and depends on the quantity of registers and shared memory used by the threads of each block. In a block, the system groups threads (typically 32) into *warps* which are executed simultaneously on successive clock cycles. The number of threads per block must be a multiple of its size to maximize efficiency. Much of the global memory latency can then be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. Consequently, the more active blocks there are per SM, and also active warps, the more the latency can be hidden. Special care must also be taken to avoid flow control instructions (if, switch, do, for, while) that may force threads of a same block to take different paths in the program and serialize the execution.

The proposed strategies are inspired by the population-based and multiple independent walks general strategies described in Section 3. However, only the LS phase is parallelized on GPU instead of entire walks. In all cases, memory management issues had to be adressed. Data transfers between the CPU and the GPU as well as global memory accesses require considerable time but may be often be reduced by storing the related data structures in shared memory. However, in the case of ILS applied to TSP, one central data structure is the distance matrix which is needed by all solutions of the population while being too large ($O(n^2)$ in size) to fit in shared memory for problems larger than a few hundred cities. It is then kept in global memory. On the other hand, as it is not modified during the LS phase, it is possible to take benefit of the texture cache to reduce their access times. The first two strategies are applied on a first-improvement LS implementation and the last one on a best-improvement scheme over a fixed size neighborhood. Specific details for each strategy are given in the next sections.

4.1 First-improvement LS : $ILS-FI_{thread}$ and $ILS-FI_{block}$

The proposed GPU parallelization strategies applied to a first-improvement LS have been presented by the authors

in the form of preliminary work [20] and are inspired by our previous contributions on Ant Colony Optimization [21], [18]. Their general parallelization model is illustrated in Figure 3(a). The first one, presented in Figure 3(b), is called $ILS-FI_{thread}$ and associates each LS to a CUDA thread. It has the advantage of allowing the execution of a great number of LSs on each SM and the drawback of limiting the use of fast GPU memory. The second strategy, called $ILS-FI_{block}$ and illustrated in Figure 3(c), associates each LS to a CUDA block. Thus, parallelism is preserved for the LS phase, but another level of parallelism is exploited by sharing the multiple neighbors between many threads of a block. As only one solution is assigned to a block, it becomes possible to store the data structures needed to improve the solution in the shared-memory. Two variants of the $ILS-FI_{block}$ strategy are then distinguished : $ILS-FI_{block}^{global}$ and $ILS-FI_{block}^{shared}$.

Currently, most efficient LS methods for solving the TSP are based on the first-improvement pivoting rule and mechanisms to reduce execution times. They were obviously designed and optimized for traditional single processor computing systems. While it may be possible to achieve good efficiency on small multiple processor/core systems with minimal changes to existing algorithms, the parallelization process is not as straightforward for a synchronous, massively parallel architecture like the GPU. On one hand, existing sequential implementations are based on reducing the number of computed neighbors in a single LS step whereas GPU processing is well suited for large data-parallel applications. On the other hand, speedup mechanims are often based on conditional statements which induce thread divergence within warps and then serialization. These observations lead us to propose a new parallelization strategy based on the synchronous evaluation of fixed size neighborhoods.

4.2 Best-improvement LS : $ILS-RKBI_{blocks}$

In a basic best-improvement LS, the whole neighborhood of a given solution is evaluated. In the case of the 3-opt move for a n cities TSP, the number of combinations to delete three edges of the tour is $\binom{n}{3}$. For each combination, there are 7 ways to reconnect the subtours [22] which corresponds to 7 neighbors. Research on TSP show that such a scheme leads to prohibitive execution times and local optima that are much further away from the ones obtained with first-improvement schemes [6]. We then propose the *random-k-best-improvement* pivoting rule which offers a compromise between first-improvement and best-improvement as well as a computation model better suited to GPUs. At each step, arcs to be deleted are randomly selected to generate a total of k neighbors and the one which produces the greatest improvement is kept. By assigning different values to k, one may customize the search behavior and the amount of work performed by the GPU. This leads to the proposition of a third parallelization strategy called $ILS-RKBI_{blocks}$ and illustrated in Figure 4(a).

This strategy splits computations of a single step into two kernels. The first one, described in Figure 4(b), is dedicated to neighborhood evaluation which is the most expensive part of the step. Each solution is associated to several blocks and its neighbors are splitted into groups to be assigned

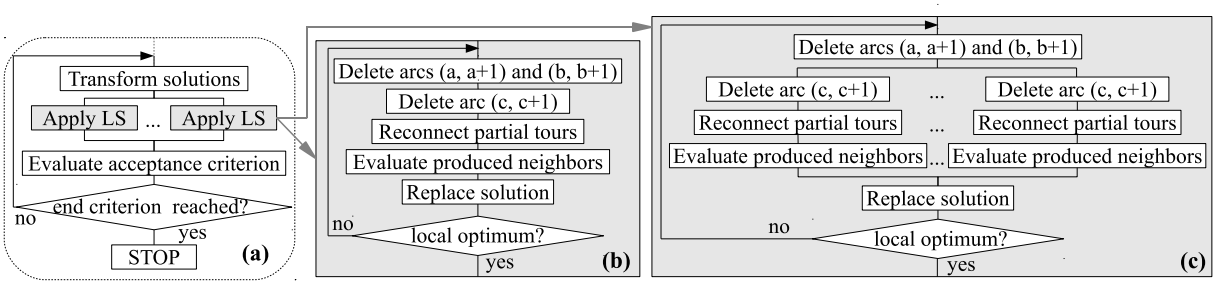

Fig. 3: Parallelization models for first-improvement ILS : general (a), $ILS-FI_{thread}$ (b) and $ILS-FI_{block}$ (c).

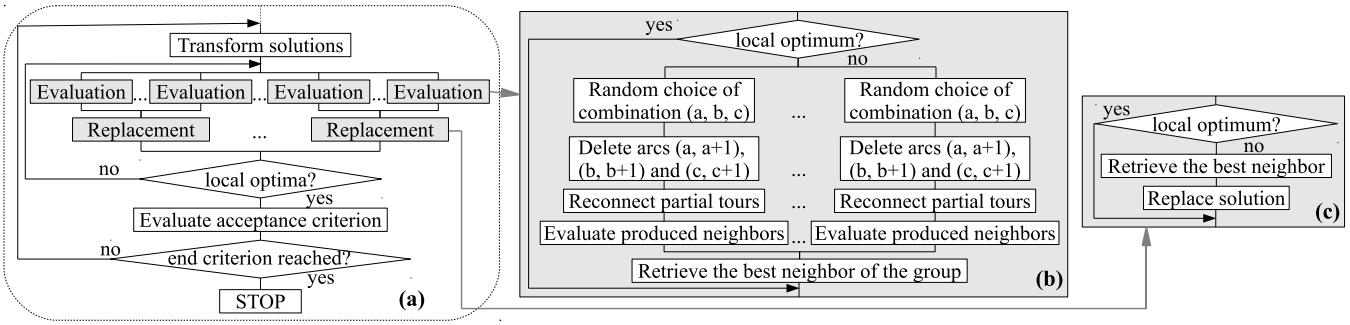

Fig. 4: Parallelization model for random-k-best-improvement ILS (a), evaluation kernel (b) and replacement kernel (c).

to each block. Each thread is then associated to several neighbors. Using the NVIDIA CURAND library, a thread randomly selects the three arcs to break, evaluates possible reconnections and stores the best one in its own space in shared memory. Once all the neighbors have been evaluated, a reduction is performed in the shared memory to find the best neighbor of the block which is stored in global memory.

In the second kernel, presented in Figure 4(c), each solution is associated to a single block which retrieves its best neighbor among the ones produced by the first kernel and replaces the current solution. The two kernels are launched alternately until all solutions are local optima.

5. Experimental results

GPU strategies are experimented on TSP problems ranging in size from 100 to 3038 cities. Speedups are computed by dividing the sequential CPU time with the parallel time obtained with the GPU acting as a co-processor. Experiments were made on a NVIDIA Fermi C2050 GPU containing 14 SMs, 32 SPs per SM and 48 KB of shared memory per SM. Code was written in the "C for CUDA V4.1" [19] programming environment. As a premiminary step, we validated our sequential ILS implementation with a comparative study with Stützle and Hoos [10] and Lourenço *et al.* [9] works.

The parallel ILS parameters are as follows. Initial solutions are built with the nearest neighbor heuristic, improved with 3-opt LS and perturbed with a double-bridge move. A population of nb_{sol} solutions is used, a total number of it_{total} iterations is performed and the ILS procedure is limited to $it_{lim} = \frac{it_{total}}{nb_{sol}}$ iterations for each solution. All speedups are computed for $nb_{sol} = 2^x$ with $x \in \{0, 3, 6, 9, 11\}$ from 20 trials for problems with less than 1000 cities and from 10 trials for larger instances.

5.1 $ILS - FI_{thread}$ and $ILS - FI_{block}$

First-improvement LS algorithms use a *don't look bits* procedure and a fixed-radius neighbor search restricted to candidate lists of size 40. it_{total} is set to 1048576 for each problem to ensure that the same number of LSs are performed for all numbers of blocks and threads. In $ILS - FI_{block}$, the number of blocks is set to nb_{sol} and the number of threads per block is set to the size of candidate lists. In $ILS - FI_{thread}$, the number of blocks and threads are configured to maximize the number of blocks without exceeding the number of active blocks per SM.

Table 1 shows speedup for $ILS - FI_{thread}$, $ILS - FI^{global}_{block}$ and $ILS - FI^{shared}_{block}$. The reader may note that increasing nb_{sol} and so, the total number of threads, leads to increasing speedups for all strategies. Overall, if the number of threads used is too small, GPU resources are not well exploited and memory latency is not efficiently hidden. The reader may also notice that speedups obtained with $ILS - FI_{thread}$ are always lower than with $ILS - FI_{block}$. As this strategy does not execute enough threads in parallel to efficiently hide memory latency, we obtain a maximal speedup of 2.02. In fact, speedups are achieved only with 2048 threads. Furthermore, code divergence induced by computing the neighbors of many solutions/threads on the same block in SIMD mode involves significant algorithm serialization.

Table 1: Speedup for $ILS - FI_{thread}$ (T), $ILS - FI^{global}_{block}$ (BG) and $ILS - FI^{shared}_{block}$ (BS) strategies and solution quality (frequency of finding the known optimum $freq$ and average percentage deviation from the optimum Δ_{avg}) for each problem.

Problem	nb_{sol}	Speedup			Solution quality	
		(T)	(BG)	(BS)	$freq$	Δ_{avg}
kroA100	1	0.01	0.08	0.08	1.00	0.000
	8	0.06	0.45	0.50	1.00	0.000
	64	0.34	2.83	3.13	1.00	0.000
	512	0.93	6.40	7.01	1.00	0.000
	2048	2.02	7.07	**7.83**	1.00	0.000
lin318	1	0.01	0.09	0.10	1.00	0.000
	8	0.06	0.51	0.57	1.00	0.000
	64	0.35	3.23	3.54	1.00	0.000
	512	0.84	6.92	7.61	1.00	0.000
	2048	1.72	7.77	**8.51**	1.00	0.000
rat783	1	0.01	0.07	0.08	1.00	0.000
	8	0.06	0.43	0.49	1.00	0.000
	64	0.34	2.69	2.18	1.00	0.000
	512	0.80	5.77	3.13	0.23	0.020
	2048	1.49	**6.37**	3.27	0.00	0.145
fl1577	1	0.01	0.07	0.08	0.33	0.182
	8	0.05	0.33	0.40	0.39	0.010
	64	0.23	1.73	0.97	0.67	0.003
	512	0.54	4.59	1.14	0.10	0.015
	2048	0.79	**5.20**	1.13	0.00	0.074
pcb3038	1	0.01	0.08	-	0.00	0.210
	8	0.06	0.46	-	0.00	0.192
	64	0.39	2.91	-	0.00	0.306
	512	0.76	5.20	-	0.00	0.667
	2048	1.27	**5.60**	-	0.00	1.104

The greater speedups and the maximal value of 7.77 obtained with $ILS - FI^{global}_{block}$ show that sharing the work associated to each solution between several threads is more efficient. For example, when nb_{sol} is set to 2048, ILS_{thread} uses 2048 threads versus 81920 for ILS_{block}. On the other hand, speedups increase from 100 to 318 cities and then slightly decrease. In that case, the larger data structures and frequent memory accesses needed to solve the biggest problems imply memory latencies that grow faster than the benefits of parallelizing available computations. Further improvements are brought by the use of shared memory, introduced in $ILS - FI^{shared}_{block}$, which provides maximal speedup of 8.51. However, results for the three biggest problems show that the limits of this kind of memory are quicky reached. In fact, as it is limited in size, using too much of it reduces the number of active blocks per SM. Associated to the combined effect of the increasing number of blocks required to perform computations, performance

gains become less significative. For the 3038 cities problem, the amount of required shared memory is so high that no block can be active on any SM.

An analysis of the frequency of finding the known optimum and the average percentage deviation from the optimum is also provided in Table 1. It shows that the optimal solution is always found by the parallel implementations for small problems. For medium-sized problems, the more nb_{sol} increases, the less frequently is the optimal solution found. As the number of iterations becomes too low to provide a thorough search, the optimal solution is never found for the bigger problem. This indicates that when choosing appropriate parameters for the parallel algorithms, a compromise must be achieved between speedup and solution quality.

5.2 $ILS - RKBI_{blocks}$ strategy

An empirical study was performed to determine the parameters used for the $ILS-RKBI_{blocks}$ strategy. The number of threads per block is set to 64 as it generally maximizes the number of active blocks per SM. It is also a multiple of 32 as advised in the CUDA specification [19]. Table 2 provides the chosen number of blocks for each solution to evaluate its neighbors. The number of iterations it_{total} and the number of evaluated neighbors k are selected so the sequential execution time is in the same order of magnitude as the ones of the first-improvement strategies when nb_{sol} is set to 64. They are then set to $1,120,000/5,500$ for kroA100, $1,120,000/11,000$ for lin318, $560,000/10,000$ for rat783, $2,240,000/1,200$ for fl1577 and $2,240,000/1,000$ for pcb3038.

Table 2: Number of blocks per solution for each problem and nb_{sol} values.

	1	8	64	512	2048
kroA100	120	70	30	5	2
lin318	110	60	30	5	2
rat783	80	100	60	10	5
fl1577	140	90	60	15	15
pcb3038	120	100	90	20	20

Table 3 presents speedup, frequency of finding the optimum and average percentage deviation from the optimum for the $ILS-RKBI_{blocks}$ strategy. The reader may notice that significant speedups are obtained with all values of nb_{sol} and a maximal speedup 45.40 is achieved with 64 solutions on the 318 cities problem. Also, with the exception of the two smallest problems with $nb_{sol} = 1$ where the amount of work is too small, speedups are in the same order of magnitude with any value of nb_{sol} for any particular problem. This shows the scalability of the neighborhood distribution strategy to different population sizes. However, the more nb_{sol} increases, greater is the deterioration of solution quality in most cases.

Table 3: Speedup and solution quality (frequency of finding the known optimum $freq$ and average percentage deviation from the optimum Δ_{avg}) for ILS_{rkbi}.

Problem	nb_{sol}	Speedup	Solution quality	
			$freq$	Δ_{avg}
kroA100	1	25.06	1.00	0.000
	8	37.21	1.00	0.000
	64	**45.29**	1.00	0.000
	512	42.12	1.00	0.000
	2048	42.32	1.00	0.000
lin318	1	27.55	0.00	0.482
	8	38.35	0.00	0.483
	64	**45.40**	0.00	0.633
	512	43.21	0.00	0.875
	2048	43.58	0.00	0.886
rat783	1	14.76	0.00	1.224
	8	16.74	0.00	1.582
	64	**17.02**	0.00	2.740
	512	16.24	0.00	3.496
	2048	16.74	0.00	3.518
fl1577	1	**30.40**	0.00	1.034
	8	29.47	0.00	1.703
	64	29.52	0.00	1.831
	512	28.02	0.00	1.776
	2048	28.85	0.00	1.741
pcb3038	1	31.23	0.00	4.636
	8	31.20	0.00	4.940
	64	31.70	0.00	5.162
	512	30.31	0.00	5.104
	2048	**33.59**	0.00	4.970

A comparison with Table 1 shows that when speedup is considered, $ILS-RKBI_{blocks}$ clearly performs better than $ILS - FI$ in all cases. The increased neighborhood size, better sharing of neighbors between blocks and reduction of thread divergence allows the resulting algorithm to make a better use of GPU ressources. Using fixed size data structures to store the best neighbors also makes the use of shared memory relevant even for bigger problems. However, the gains in speedup also lead to the deterioration of solution quality. The random choice of neighbors and inhability to use speedup mechanisms of first-improvement strategies makes it difficult for $ILS - RKBI_{blocks}$ to keep the same level of optimization than $ILS - FI_{thread}$ and $ILS - FI_{block}$ in the same execution time. This may indicate either that a compromise must be found between speedup and solution quality when designing parallel ILS algorithms for GPUs or that news ways must be thought to restore - if possible - the delicate balance between all the algorithm components in this context.

6. Conclusion

The aim of this paper was to design efficient parallelization strategies for Iterated Local Search on Graphics Processing Units to solve the Travelling Salesman Problem. The $ILS-FI_{thread}$ and $ILS-FI_{block}$ strategies associated the local search phase to the execution of streaming processors and multiprocessors respectively. They provided maximal

speedups of 2.02 and 8.51 with competitive solution quality as well as major shortcomings in their use of GPU computational resources. In an attempt to overcome these limitations, the $ILS - RKBI_{blocks}$ strategy was proposed to increase neighborhood size and associate multiple blocks to the evaluation of each solution. Significant speedups were then achieved, ranging as high as 45.40, at the cost of a variable deterioration of solution quality. This leads to the idea that solving the TSP and other combinatorial optimization problems on GPU is currently a matter of compromise between speedup and search efficiency.

In future works, we plan to deepen our understanding of the links between neighborhood size, pivoting rules and algorithm parameters in order to improve the search process of our GPU ILS implementation. Also, as this work is part of a greater project related to the parallelization of metaheuristics on GPU, we seek to use the knowledge built in this paper to propose a general framework that can be applied to other metaheuristics. We also plan to propose algorithms to automatically determine effective thread/block/GPU configurations for ILS and other metaheuristics. We believe that the global acceptance of GPUs as components for optimization systems requires algorithms and software that are not only effective, but also usable by a wide range of academicians and practitioners.

Acknowledgment

This work has been supported by the Agence Nationale de la Recherche (ANR) under grant no. ANR-2010-COSI-003-03. The authors would also like to thank the Centre de Calcul Régional Champagne-Ardenne for the availability of the computational resources used for experiments.

References

[1] H. Hoos and T. Stützle, *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann, Elsevier, 2004.

[2] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, pp. 2245–2269, 1965.

[3] G. A. Croes, "A method for solving traveling salesman problems," *Operations Research*, vol. 6, pp. 791–812, 1958.

[4] S. Lin and B. Kernighan, "An effective heuristic algorithm for the travelling salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[5] M. Yannakakis, "The analysis of local search problems and their heuristics," in *STACS*, ser. Lecture Notes in Computer Science, vol. 415. Springer, 1990, pp. 298–311.

[6] E. Anderson, "Mechanisms for local search," *European Journal of Operational Research*, vol. 88, no. 1, pp. 139–151, 1996.

[7] J. Bentley, "Fast algorithms for geometric traveling salesman problems," *ORSA Journal on Computing*, vol. 4, no. 4, pp. 387–411, 1992.

[8] D. Johnson and L. McGeoch, *The Travelling Salesman Problem: A Case Study in Local Optimization*, ser. E.H.L. Aarts and J.K. Lenstra, editors, Local Search in Combinatorial Optimization. John Wiley & Sons, 1997, pp. 215–310.

[9] H. Lourenço, O. Martin, and T. Stützle, *Iterated local search: framework and applications*, ser. Handbook of metaheuristics. Springer, 2010, pp. 363–397.

[10] T. Stützle and H. Hoos, *Analysing the run-time behaviour of iterated local search for the traveling salesman problem*, ser. Essays and surveys in metaheuristics. Springer, 2001, pp. 21–43.

[11] M. Verhoeven and E. Aarts, "Parallel local search," *Journal of Heuristics*, vol. 1, pp. 43–65, 1995.

[12] I. Hong, A. Kahng, and B. Moon, "Improved large-step markov chain variants for the symmetric tsp," *Journal of Heuristics*, vol. 3, pp. 63–81, September 1997.

[13] O. Martin and S. Otto, "Combining simulated annealing with local search heuristics," *Annals of Operations Research*, vol. 63, pp. 57–75, 1996.

[14] T. Luong, N. Melab, and E. Talbi, "Neighborhood structures for gpu-based local search algorithms," *Parallel Processing Letters*, vol. 20, no. 4, pp. 307–324, 2010.

[15] ——, "Gpu computing for parallel local search metaheuristic algorithms," *IEEE Transactions on Computers*, vol. 99, no. PrePrints, 2011.

[16] M. O'Neil, D. Tamir, and M. Burtscher, "A parallel gpu version of the traveling salesman problem," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011, pp. 348–353.

[17] N. Fujimoto and S. Tsutsui, "A highly-parallel tsp solver for a gpu computing platform," in *NMA*, 2010, pp. 264–271.

[18] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *Journal of Parallel and Distributed Computing*, 2012.

[19] *CUDA : Computer Unified Device Architecture Programming Guide 4.1*, 2012. [Online]. Available: http://www.nvidia.com

[20] A. Delévacq, P. Delisle, and M. Krajecki, "Parallel gpu implementation of iterated local search for the travelling salesman problem," in *Learning and Intelligent OptimizatioN*. Lecture Notes in Computer Science, 2012.

[21] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," in *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, H. R. Arabnia, S. C. Chiu, G. A. Gravvanis, M. Ito, K. Joe, H. Nishikawa, and A. M. G. Solo, Eds. CSREA Press, 2010, pp. 196–202.

[22] G. Gutin and A. Punnen, *The Traveling Salesman Problem and Its Variations*, ser. Combinatorial Optimization. Kluwer Academic Publishers, 2002.