Parallel GPU Implementation of Iterated Local Search for the Travelling Salesman Problem

Audrey Delévacq, Pierre Delisle, and Michaël Krajecki

CReSTIC, Université de Reims Champagne-Ardenne, Reims, France
{audrey.delevacq,pierre.delisle,michael.krajecki}@univ-reims.fr

Abstract. The purpose of this paper is to propose effective parallelization strategies for the Iterated Local Search (ILS) metaheuristic on Graphics Processing Units (GPU). We consider the decomposition of the 3-opt Local Search procedure on the GPU processing hardware and memory structure. Two resulting algorithms are evaluated and compared on both speedup and solution quality on a state-of-the-art Fermi GPU architecture. We report speedups of up to 6.02 with solution quality similar to the original sequential implementation on instances of the Travelling Salesman Problem ranging from 100 to 3038 cities.

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

1 Introduction

Iterated Local Search (ILS) is a metaheuristic that successively applies a Local Search (LS) procedure to an initial solution and incorporates mechanisms to climb out of local optima. It finds good solutions to many optimization problems in a reasonable time which may remain too high in practice. Even though this time can be reduced by parallel computing, most approaches are dedicated to CPU-based architectures. As research on computer architectures is rapidly evolving, new types of hardware have recently become available. Among them are Graphics Processing Units (GPU) which provide great and affordable computing power but also require new algorithmic paradigms to be used efficiently.

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. We first present k-opt LS algorithms and the ILS metaheuristic. Then, after a literature review on parallel LS and ILS, the proposed GPU parallelization strategies are detailed and experimented.

2 Iterated Local Search for the TSP

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

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. 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. Figure 1 describes the specific 3-opt method [5]. As a LS procedure may become trapped in a local optimum, it is often embedded in a guiding construction such as Iterated Local Search (ILS) [6]. Figure 2 shows the main steps of this metaheuristic.

```
Compute length L of solution S
while S is improved do
    for all a, b, c ∈ [0; n] do
        Delete arcs (a,a+1), (b,b+1) and (c,c+1)
        Produce S' by reconnecting partial tours
         with other arcs
        Compute length L' of solution S'
        if L' < L then
            S = S' and L = L'
Return best solution S
```

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

```
Generate solution S
Apply LS procedure on S
Compute 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'
    Compute length L' of solution S'
    if L' < L then //acceptance criterion
        S = S' and L = L'
Return best solution S
```

Fig. 2. ILS pseudo-code.

The works of Stützle and Hoos [10] and Lourenço *et al.* [6] show the efficiency of ILS in solving 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 and ILS

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 [4] 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.* [3] designed a parallel ILS which executes a total of m iterations

using a pool of p solutions. Martin and Otto [9] proposed an implementation in which several solutions are computed simultaneously on different processors and the best solution replaces all solutions at irregular intervals.

Luong *et al.* [8] proposed a methodology for implementing large neighborhood LS algorithms on GPU. The CPU is in charge of all LS processes while the GPU generates and evaluates neighbor solutions which are associated to CUDA threads. This methodology is experimented with Tabu Search on the Permuted Perceptron Problem and maximal speedup of 25.8 is reported. In Luong *et al.* [7], the GPU Tabu Search is embedded with ILS to solve the Quadratic 3-dimensional Assignement Problem with maximal speedup of 6.1.

Most works related to parallel LS and ILS are based on traditional CPU architectures. As LS algorithms are key underlying components of most high-performing metaheuristics, a natural fit would be 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 and technical work to achieve in order to design such hybrid parallel combinatorial optimization methods. This paper aims to partially fill this gap by proposing and evaluating GPU implementations of an ILS algorithm for the TSP based on 3-opt parallel LS.

4 Parallel GPU strategies for ILS

We present two GPU strategies for ILS which mainly differ by the way they distribute solutions to processing elements and by their use of GPU shared memory. Beforehand, we provide a brief description of the GPU computing environment.

The NVIDIA GPU [1] architecture includes many *Streaming Multiprocessors* (SM), each one of them being composed of *Streaming Processors* (SP). Each SM allows the execution of many threads in a data-parallel fashion. 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. *Constant* and *texture memory caches* provide faster access to device memory but are read-only. 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. In the CUDA programming model [1], the GPU works as a co-processor of a conventional CPU. It allows the parallel execution of many CUDA threads that are grouped into *blocks* to be executed by the SMs. However, the number of blocks that a SM can process at the same time (*active blocks*) is restricted by the available shared memory and registers. 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 ILS implementations are inspired by the multiple independent walks strategy described in Section 3. However, only the LS is parallelized on GPU instead of entire walks. In this scheme, illustrated in Figure 3(a), LS is applied on each solution on different processing elements.

On a conventional CPU architecture, the concept of processing element is usually associated to a single-core processor or to one of the cores of a multi-

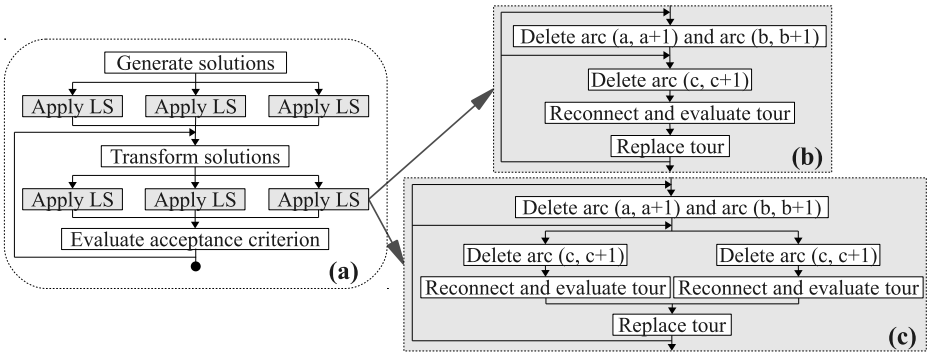

Fig. 3. Parallelization models for ILS : general (a), ILS_{thread} (b) and ILS_{block} (c).

core processor. On a GPU, the obvious choice is to associate this concept to a single SP. In that case, a first strategy that may be defined is to associate each LS to a CUDA thread. Each thread then improves its solution in a SIMD fashion. This strategy, called ILS_{thread}, is illustrated in Figure 3(b) and has been proposed for the parallelization of Ant Colony Optimization in previous work by the authors [2]. 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_{block} and illustrated in Figure 3(c), is based on associating the concept of processing element to a whole SM. In that case, each solution is associated to a CUDA block and parallelism is preserved for the LS phase. A single thread of a given block is still in charge of applying LS to a solution, but another level of parallelism is exploited by sharing the multiple neighbors between many threads of a block. Following the idea of ILS_{thread}, a simple implementation would then imply keeping the private data structures of a solution in the global memory. However, as only one solution is assigned to a block and so to a SM, it becomes possible to store the data structures needed to improve the solution in the shared-memory. Two variants of the ILS_{block} strategy are then distinguished and experimented : ILS_{block}^{global} and ILS_{block}^{shared}.

5 Experimental results

The proposed GPU strategies for ILS are experimented on TSP problems with sizes varying from 100 to 3038 cities. Speedups are computed by dividing the sequential CPU time with the parallel time, which is obtained with the same CPU and 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 V4.0" [1] programming environment. As a premiminary step, we validated our sequential ILS with a comparative study with Stützle and Hoos [10] and Lourenço *et al.* [6] works.

The parallel ILS algorithm parameters are as follows. A population of nb_{sol} solutions is used, a total number of 1048576 iterations is performed and the ILS

procedure is limited to $it_{lim} = \frac{1048576}{nb_{sol}}$ iterations for each solution. All speedups are computed for $nb_{sol} = 2^x$ with $x \in \{8, 9, 10, 11\}$ from 20 trials for problems with less than 1000 cities and from 10 trials for larger instances.

Figure 4 shows the speedups obtained for each problem and each parallelization strategy. The reader may note that increasing the number of solutions nb_{sol} and so, the total number of threads, generally leads to increasing speedups for all strategies. Moreover, speedups obtained with ILS_{thread} are limited to 2.02 and are always lower than with ILS_{block}. This strategy does not execute enough threads in parallel to efficiently hide memory latency. Furthermore, code divergence induced by computing the neighbors of many solutions/threads on the same block in SIMD mode involves significant algorithm serialization.

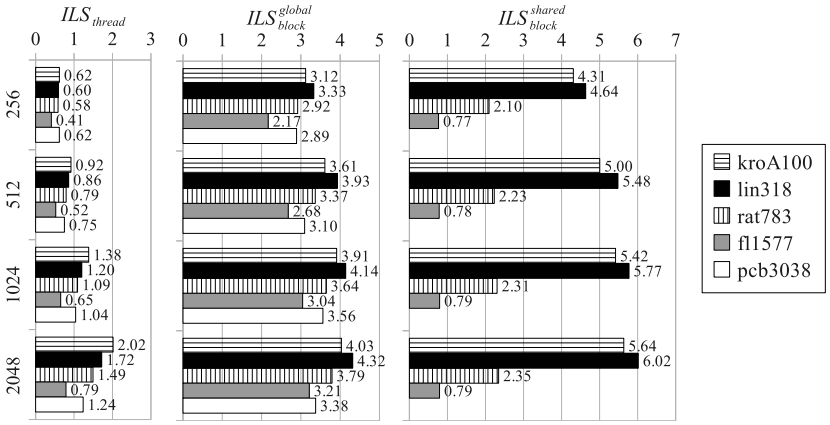

Fig. 4. Speedups for ILS_{thread}, ILS^{global}_{block} and ILS^{shared}_{block} strategies for each nb_{sol}.

The greater speedups and the maximal value of 4.32 obtained with ILS^{global}_{block} show that sharing the work associated to each solution between several threads is more efficient. However, speedups increase from 100 to 318 cities and then slightly decrease. In that case, the larger data structures and frequent memory accesses imply memory latencies that grow faster than the benefits of parallelizing available computations. Further improvements are brought by the use of shared memory of ILS^{shared}_{block}, which provides a maximal speedup of 6.02. However, results for the three biggest problems show that the limits of this kind of memory are quickly reached. In fact, since this memory is very limited in size, either speedup is not achieved or the problem can not be solved at all.

An analysis of the average percentage deviation Δ from the optimum showed that the optimal solution is always found ($\Delta = 0.000$) by the parallel implementations for small problems. For medium-sized problems, the more nb_{sol} increases, the less frequently the optimal solution is found ($\Delta = 0.002$ to $\Delta = 0.137$). As the number of iterations becomes too low to provide a thorough search, the optimal solution is never found for the bigger problem ($\Delta = 0.520$ to $\Delta = 1.112$). This indicates that when choosing appropriate parameters for the parallel algorithms, a compromise must be achieved between speedup and solution quality.

6 Conclusion

The aim of this paper was to design efficient parallelization strategies for the implementation of Iterated Local Search on Graphics Processing Units to solve the Travelling Salesman Problem. The ILS_{thread} and ILS_{block} strategies associated the Local Search phase to the execution of streaming processors and multiprocessors respectively. Experimental results showed significant speedups of up to 6.02 with solution quality often equal or close to optima, but also considerable limitations on large problems. Moreover, they highlighted that maximal exploitation of GPU ressources often requires algorithmic configurations that do not let ILS perform an effective exploration and exploitation of the search space.

In future work, we plan to study the GPU performance of other decomposition approaches like tour-based partitioning. We would also like to design k-opt based parallel algorithms that provide a better compromise between GPU efficiency and search robustness for the TSP and related problems.

Acknowledgments. 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. CUDA : Computer Unified Device Architecture Programming Guide 4.0, http://www.nvidia.com (2011)
2. Delévacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. In: PDPTA'10. pp. 196–202. CSREA Press (2010)
3. Hong, I., Kahng, A., Moon, B.: Improved large-step markov chain variants for the symmetric tsp. Journal of Heuristics 3, 63–81 (September 1997)
4. Johnson, D., McGeoch, L.: The Travelling Salesman Problem: A Case Study in Local Optimization, pp. 215–310. E.H.L. Aarts and J.K. Lenstra, editors, Local Search in Combinatorial Optimization, John Wiley & Sons (1997)
5. Lin, S.: Computer solutions of the traveling salesman problem. Bell System Technical Journal 44, 2245–2269 (1965)
6. Lourenço, H., Martin, O., Stützle, T.: Iterated local search: framework and applications, pp. 363–397. Handbook of metaheuristics, Springer (2010)
7. Luong, T., Loukil, L., Melab, N., Talbi, E.: A gpu-based iterated tabu search for solving the quadratic 3-dimensional assignment problem. In: AICCSA. pp. 1–8 (2010)
8. Luong, T., Melab, N., Talbi, E.: Neighborhood structures for gpu-based local search algorithms. Parallel Processing Letters 20(4), 307–324 (2010)
9. Martin, O., Otto, S.: Combining simulated annealing with local search heuristics. Annals of Operations Research 63, 57–75 (1996)
10. Stützle, T., Hoos, H.: Analysing the run-time behaviour of iterated local search for the traveling salesman problem, pp. 21–43. Essays and surveys in metaheuristics, Springer (2001)
11. Verhoeven, M., Aarts, E.: Parallel local search. Journal of Heuristics 1, 43–65 (1995)