J. Parallel Distrib. Comput. 73 (2013) 52–61

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel Ant Colony Optimization on Graphics Processing Units

Audrey Delévacq [a], Pierre Delisle [a,*], Marc Gravel [b], Michaël Krajecki [a]

[a] CReSTIC, Université de Reims Champagne-Ardenne, Reims, 51687, France

[b] Département d'Informatique et de Mathématique, Université du Québec à Chicoutimi, Saguenay, Canada

ARTICLE INFO

Article history:
Received 9 June 2011
Received in revised form
15 December 2011
Accepted 10 January 2012
Available online 25 January 2012

Keywords:
Ant colony optimization
Parallel metaheuristics
GPU
CUDA
MMAS
Parallel ants
Multiple colonies

ABSTRACT

The purpose of this paper is to propose effective parallelization strategies for the Ant Colony Optimization (ACO) metaheuristic on Graphics Processing Units (GPUs). The Max–Min Ant System (MMAS) algorithm augmented with 3-opt local search is used as a framework for the implementation of the *parallel ants* and *multiple ant colonies* general parallelization approaches. The four resulting GPU algorithms are extensively evaluated and compared on both speedup and solution quality on a state-of-the-art Fermi GPU architecture. A rigorous effort is made to keep parallel algorithms true to the original MMAS applied to the Traveling Salesman Problem. We report speedups of up to 23.60 with solution quality similar to the original sequential implementation. With the intent of providing a parallelization framework for ACO on GPUs, a comparative experimental study highlights the performance impact of ACO parameters, GPU technical configuration, memory structures and parallelization granularity.

1. Introduction

The Ant Colony Optimization (ACO) metaheuristic [17] is a constructive, population-based approach based on the social behavior of ants. As it is acknowledged as a powerful method to solve combinatorial optimization problems, a considerable amount of work is dedicated to improving its performance. Among the proposed solutions, we find the use of parallel computing to reduce computation time, improve solution quality or both.

Most parallel ACO implementations can be classified into two general approaches. The first one is the parallel execution of the ants construction phase in a single colony. Initiated by Bullnheimer et al. [5], it aims to accelerate computations by distributing ants to computing elements. The second one, introduced by Stützle [27], is the execution of multiple ant colonies. In this case, entire ant colonies are attributed to processors in order to speedup computations as well as to potentially improve solution quality by introducing cooperation schemes between colonies. These implementations usually follow the *message-passing* and *shared-memory* computing paradigms. The relatively high-level abstraction model they provide facilitates the development of effective and portable optimization software on conventional CPU-based parallel architectures.

However, as research on parallel architectures is rapidly evolving, new types of hardware have recently become available for high performance computing. Among them, we find Graphics Processing Units (GPUs) which provide great computing power at an affordable cost but are difficult to program. In fact, it is not clear that conventional paradigms are suitable for expressing parallelism in a way that is efficiently implementable on GPU architectures. As academic and industrial combinatorial optimization problems always increase in size and complexity, the field of parallel metaheuristics has to follow this evolution of high performance computing.

The purpose of this paper is to propose parallel implementations of ACO that are suitable for GPU computing environments. For both *parallel ants* and *multiple colonies* general approaches, two parallelization strategies are designed and experimentally compared on speedup and solution quality. Important algorithmic, technical and programming issues are also addressed in this context.

This paper is organized as follows. First, we present the ACO metaheuristic, the Max–Min Ant System (MMAS) algorithm and its application to the Traveling Salesman Problem (TSP). We choose MMAS and TSP to focus on algorithmic aspects of ACO and technical issues of GPU computing that are not problem dependent, as well as to strictly compare our results to the original works of Stützle and Hoos [28]. After a fairly complete review of the literature on parallel ACO, the proposed GPU parallelization strategies for

* Corresponding author.
E-mail addresses: audrey.delevacq@univ-reims.fr (A. Delévacq), pierre.delisle@univ-reims.fr (P. Delisle), mgravel@uqac.ca (M. Gravel), michael.krajecki@univ-reims.fr (M. Krajecki).

MMAS are explained. Finally, extensive experimental results are presented to evaluate and compare their performance.

2. Ant Colony Optimization for the traveling salesman problem

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

In most ACO algorithms, a given number of ants gradually and concurrently build tours using heuristic and pheromone information. Ants update pheromone values in the process to guide other ants to potentially better tours. In many cases, updates are performed according to the tours built and to various general rules. Following a given number of iterations where ants have built many – hopefully – improved solutions, the best one is chosen as the solution of the problem. A complete description of ACO can be found in Dorigo [16,17].

The Max–Min Ant System (MMAS) [28] is generally recognized as one of the most effective ACO algorithms at the present time. It also incorporates the main mechanisms and memory structures that are common to most algorithmic versions of this metaheuristic. Fig. 1 illustrates a simplified pseudo-code of the MMAS. In this algorithm, the number of ants m is set to the number of cities n. The ants tour construction process is performed in each of the ni iterations. To that end, each ant ant_k is initially placed on a randomly chosen city. Then, at each solution construction step, ant_k builds its tour T_k by repeatedly applying a state transition rule to choose the cities that will be added to its tour among the unvisited cities. After all ants have built their tour, pheromone τ is updated according to some rule which follows two objectives: to increase the desirability of arcs associated to the global best solution found so far T_{gl} or to the best solution of the current iteration T_{it} and to reduce the ones that have not been used. To avoid search stagnation, τ is kept between minimal and maximal values $\tau_{\min}$ and $\tau_{\max}$ on each arc. τ is also initialized at $\tau_{\max}$ in order to facilitate exploration of the search space in the beginning of the algorithm. Moreover, MMAS uses a trail-smoothing mechanism which also promotes exploration by increasing the probability of choosing arcs with low pheromone values.

For better readability, the state transition and pheromone update rules are not explained in this paper. More information on these subjects may be found in the original works of Stützle and Hoos [28]. However, it is important to mention that the state transition rule computes the probability for each unvisited city to be chosen by the ant according to distance and pheromone values. These are stored in two $n \times n$ matrices that need to be available to each ant. Consequently, when implementing MMAS (as well as most ACO algorithms) on a real computer architecture, memory must be large enough to accommodate these data structures and fast enough to keep up with the numerous requests from processing elements. This requirement becomes more prohibitive as problem size increases.

When faced with large problems, MMAS uses candidate lists to reduce the possible cities to be chosen by ants during the tour construction phase. These lists contain, for each city, a given number of its cl nearest neighbors sorted in increasing order. Ants choose cities exclusively in candidate lists until all candidates are visited. Only in that case is an ant allowed to pick a city outside the lists.

Finally, MMAS may be augmented with a local search procedure such as 3-opt [21] to improve the solutions found by the ants. This

```
Initialize colony parameters
Initialize pheromone matrix $\tau$ with $\tau_{max}$ for each pair of cities
for $t = 1$ to $ni$ do
    for $k = 1$ to $m$ do {TOUR CONSTRUCTION}
        Place $ant_k$ on a randomly chosen city
        while the $n$ cities are not all visited do
            Move $ant_k$ to next city according to state transition rule
        Compute length $L_k$ of the tour $T_k$ produced by $ant_k$
    for $k = 1$ to $m$ do {LOCAL SEARCH}
        while $T_k$ is improved do
            for all $0 < a < n$ and $0 < b < n$ and $0 < c < n$ do
                Delete arcs $(a, a + 1), (b, b + 1), (c, c + 1)$
                Produce $T'$ by reconnecting partial tours with other arcs
                Compute $L'$
                if $L' < L_k$ then
                    Update $T_k$ with $T'$
    for $k = 1$ to $m$ do
        if $L_k < L_{it}$ then
            Update $T_{it}$ with $T_k$
    if $L_{it} < L_{gl}$ then
        Update $T_{gl}$ with $T_{it}$
    Evaporate $\tau$ and update $\tau$ for each pair of cities of $T_{it}$ or $T_{gl}$
    Control $\tau_{min} < \tau < \tau_{max}$ and update $\tau$ with trail smoothing mechanism
```

Fig. 1. MMAS pseudo-code with local search.

method aims to improve a current solution by replacing at most three of its arcs. The process of replacing the current solution with an improved one is then iterated until no better solution is found.

ACO algorithms have proven to be successful in solving many academic and industrial combinatorial optimization problems [17]. However, faced with large and hard problems, they need a considerable amount of computing time and memory space to be effective in their exploration of the search space. Consequently, some interest in their parallelization has been raised in the recent years. The following section presents a literature review on parallel ACO.

3. Literature review on parallel Ant Colony Optimization

The concurrent nature of both tour construction and global search of the solution space makes the ACO metaheuristic a good candidate for parallelization. However, this potential comes with important challenges mainly due to pheromone management and to the size of the data structures that have to be maintained. Works on traditional, CPU-based parallel ACO can be classified into two general approaches: *parallel ants* and *multiple ant colonies*. These approaches are briefly explained in Sections 3.1 and 3.2. On the other hand, few authors have proposed parallel implementations dedicated to specific architectures. Section 3.3 is dedicated to these *hardware-oriented* approaches. In all cases, a survey of related works is also provided.

3.1. Parallel ants

Works related to the parallel ants approach, which aims to execute the ants tour construction phase on many processing elements, were initiated by Bullnheimer et al. [5]. They proposed two parallelization strategies for the Ant System on a message passing and distributed-memory architecture. The first one is a low-level and synchronous strategy that aims to accelerate computations by distributing ants to processors in a master-slave fashion. At each iteration, the master broadcasts the pheromone structure to slaves, which then compute their tours in parallel and send them back to the master. The time needed for these global communications and synchronizations implies a considerable overhead. The second strategy aims to reduce it by letting the algorithm perform a given number of iterations without exchanging information. The authors conclude that this partially asynchronous strategy is preferable due to the considerable reduction of the communication overhead.

The works of Talbi et al. [29], Randall and Lewis [24], Islam et al. [19], Craus and Rudeanu [8], Stützle [27] and Doerner et al. [15] are based on a similar parallelization approach and a distributed memory architecture. Delisle et al. [13,11] implemented this scheme on shared-memory architectures like SMP computers and multi-core processors. They also compared performance between the two types of architectures [12].

3.2. Multiple ant colonies

The multiple ant colonies approach, also based on a message-passing and distributed memory architecture, aims to execute whole ant colonies on available processing elements. It was introduced by Stützle [27] with the parallel execution of multiple independent copies of the same algorithm. Middendorf et al. [23] extended this approach by introducing four information exchange strategies between ant colonies: exchange of globally best solution, circular exchange of locally best solutions, migrants or locally best solutions plus migrants. It is shown that it can be advantageous for ant colonies to avoid communicating too much information and too often. Giving up on the idea of sharing whole pheromone information, they based their strategy on the trade of a single solution at each exchange step.

Chu et al. [7], Manfrin et al. [22], Ellabib et al. [18] and Alba et al. [2] have also proposed different information exchange strategies for the multiple ant colony approach. Many parameters are studied like the topology of the links between processors as well as the nature and frequency of information exchanges. These strategies are implemented using MPI on distributed memory architectures. On the other hand, Delisle et al. [10] adapted some of them on shared-memory architectures.

3.3. Hardware-oriented parallel ACO

Even though they mostly follow the parallel ants and multiple ant colonies approaches, hardware-oriented approaches are dedicated to specific and untraditional parallel architectures. Scheuermann et al. [26,25] designed parallel implementations of ACO on Field Programmable Gate Arrays (FPGA). Considerable changes to the algorithmic structure of the metaheuristic were needed to take benefit of this particular architecture.

Few authors have tackled the problem of parallelizing ACO on GPU in the form of preliminary work. Catala et al. [6] propose an implementation of ACO to solve the Orienteering Problem. Instances of up to a few thousand nodes are solved by building solutions on GPU. Wang et al. [30] propose an implementation of the MMAS where the tour construction phase is executed on a GPU to solve a 30 city TSP. You [31] provides some results obtained with a similar implementation of the Ant System, reporting speedups ranging from 2 to 20 approximately on TSPs containing from 50 to 800 cities approximately. Zhu and Curry [33] implement the same strategy on an ACO with pattern search for nonlinear function optimization problems. Speedups ranging from 128 to 403 are provided as computational results using 15,360 ants. We also report the works of Li et al. [20] on a fine-grained ACO implementation. Finally, Delévacq et al. [14] have proposed an implementation of an Ant System and a comparative study to show the influence of various ACO and GPU parameters on performance.

Bai et al. [3] provides a multi-colony implementation of MMAS where each colony has different parameters. For each colony, the whole execution of each iteration is deported on the GPU. Speedups between 2.2 and 2.3 are reported on TSPs with sizes varying from 51 to 400 cities.

These preliminary works provide some insights on the level of performance that can be experimentally achieved with different GPU implementations of ACO. However, they also present major shortcomings that undermine their scientific credibility and leave many important questions unanswered. First, speedups are mostly provided with nonexistent or inappropriate evaluation of solution quality as well as unrealistic or uncommon parameter settings. Also, strategies based on associating each ant to a single thread are proposed without any alternative approaches and multi-colony strategies that perform well on both speedup and solution quality are missing. Moreover, methods are often simplified to become uncompetitive with state-of-the-art ACO metaheuristics or experimented on TSPs limited to a few hundred cities without bypassing obvious memory limits of actual GPUs. Finally, the effects of using the different memory structures of GPUs on actual parallel ACO performance are not understood well.

As the use of GPUs for general purpose computing is emerging, the field of parallel metaheuristics has to follow this evolution. However, there is still much conceptual, technical and comparative work to achieve in order to effectively exploit this massively parallel and affordable architecture for combinatorial optimization. This is especially true in the case of ACO where extensive and rigorous works are still missing. This paper aims to partially fill this gap by proposing, evaluating and comparing various GPU implementations of a state-of-the-art ACO algorithm for the TSP: the Max–Min Ant System as it has been defined by Stützle and Hoos [28]. An effort is made to keep the parallel algorithms true to the original sequential one and to avoid degrading solution quality for the sake of improving speedups. This is shown by choosing the TSP benchmarks accordingly and by comparing the results of the MMAS both with and without local search, as it was done in the original paper.

For both parallel ants and multiple colonies approaches, two GPU parallelization strategies are proposed and compared. Many GPU and memory configurations are also evaluated. The proposed work is incremental in the sense that each strategy is built upon knowledge provided by the results of previous ones. Moreover, in order to better exploit the GPU architecture, a strategy of finer grain than parallel ants is proposed: computation of the transition rule in parallel. We also show that this strategy can be adapted to the standard 3-opt local search procedure used by the ants.

A last contribution of this article is to provide results on a state-of-the-art GPU architecture, the NVIDIA Fermi, on problem sizes to up to 2103 cities. This brings insight not only on the results of solving considerably larger TSPs than the ones found in the literature, but also on the actual limits of the GPU approach for conventional ACO algorithms.

Next section presents the proposed parallelization strategies for MMAS on GPU architectures.

4. Parallel GPU strategies for MMAS

As explained in Section 3, parallelization of ACO algorithms usually follow the general paradigms of parallel ants in a single colony and multiple ant colonies. Ants in the former case, or colonies in the latter case, are distributed to processing elements. Sections 4.1–4.4 are dedicated to the adaptation of these paradigms to the GPU architecture. In each case, two implementation strategies are proposed. They mainly differ by their definition of processing elements and by their use of GPU memories. Beforehand, for the sake of completeness, a brief description of the GPU architecture and computational model are given.

4.1. GPU architecture and CUDA programming model

The conventional NVIDIA GPU [9] includes many *Streaming Multiprocessors* (SM), each one of them being composed of *Streaming Processors* (SP). Several memories are distinguished on this special hardware, differing in size, latency and access type (read-only or read/write).

Device memory is relatively large in size but slow in access time. The *global* and *local* memory spaces are specific regions of the device memory that can be accessed in read and write modes. Data structures of a computer program to be executed on GPU must be created on the CPU and transferred on global memory which is accessible to all SPs of the GPU. On the other hand, local memory stores automatic data structures that consume more registers than available.

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. It is composed of a *constant memory cache*, a *texture memory cache*, a *shared memory* and *registers*. Constant and texture caches are linked to the constant and texture memories that are physically located in the device memory. Consequently, they are accessible in read-only mode by the SPs and faster in access time than the rest of the device memory. 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 local shared memory, which is fast in access time but small in size. It is divided into memory banks of 32-bits words that can be accessed simultaneously. This implies that parallel requests for memory addresses that fall into the same memory bank cause the serialization of accesses [9]. Registers are the fastest memories available on a GPU but involve the use of slow local memory when too many are used. Moreover, accesses may be delayed due to register read-after-write dependences and register memory bank conflicts.

GPUs are programmable through different Application Programming Interfaces like CUDA, OpenCl or DirectX. However, as current general-purpose APIs are still closely tied to specific GPU models, we choose CUDA to fully exploit the available state-of-the-art NVIDIA Fermi architecture. In the CUDA programming model [9], the GPU works as a SIMT 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. Threads within a block can cooperate by sharing data through the shared memory and by synchronizing their execution to coordinate memory accesses. 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.

It is important to note that in the context of GPU execution, flow control instructions (if, switch, do, for, while) can affect the efficiency of an algorithm. In fact, depending on the provided data, these instructions may force threads of a same warp to diverge, in other words, to take different paths in the program. In that case, execution paths must be serialized, increasing the total number of instructions executed by this warp.

4.2. Parallel ants

In the parallel ants general strategy, ants of a single colony are distributed to processing elements in order to execute tour constructions in parallel. 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-core processor. On a GPU architecture, as previous work on ACO has shown, 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 ant to a CUDA thread. Each thread then computes the state transition rule of each ant in a SIMD fashion. We call this strategy ANT_{thread}. It has the advantage of allowing the execution of a great number of ants on each SM and the drawback of limiting the use of fast GPU memory. In fact, each ant needs its own data structures, mainly tour and probability arrays (of size $O(n)$), to effectively compute the state transition rule required to build a solution. Simple calculations show that using the shared memory for these structures would restrict the algorithm to use a very small number of ants on a single SM and that this restriction would grow linearly with problem size. Code optimizations may help raise that number by a constant factor, but hardly enough to bypass algorithmic limitations. Therefore, these data structures must be stored in global memory and accessed in read/write mode during the tour construction phase.

The second proposed strategy is based on associating the concept of processing element to a whole SM. In that case, each ant is associated to a CUDA block and parallelism is preserved for tour construction. We call this strategy ANT_{block}. A single thread of a given block is still in charge of executing the tour construction of an ant, but an additional level of parallelism may be exploited in the computation of the state transition rule. In fact, an ant evaluates several candidate cities before selecting the one to add to its current solution. As these evaluations can be done in parallel, they are assigned to the remaining threads of the block.

Following the idea of the first strategy, a simple implementation would then imply keeping ant's private data structures in the global memory. However, as only one ant is assigned to a block and so to a SM, taking advantage of the shared-memory becomes possible for problems bigger than a few dozen cities. Data needed to compute the ant state transition rule is then stored in this memory that is faster and accessible by all threads that participate in the computation. In order to evaluate the benefits and limits of using the shared-memory in this context, two variants of the ANT_{block} strategy are distinguished: ANT_{block}^{global} and ANT_{block}^{shared}.

Most remaining issues encountered in the GPU implementation of the parallel ants general strategy are related to memory management. More particularly, data transfers between CPU and GPU as well as global memory accesses require considerable time. As it was mentioned before, these accesses may be reduced by storing the related data structures in shared memory. However, in the case of ACO, the three central data structures are the pheromone matrix, the distance matrix and the candidates lists, which are needed by all ants of the colony while being too large (ranging from $O(n * cl)$ to $O(n^2)$ in size) to fit in shared memory. They are then kept in global memory. On the other hand, as they are not modified during the tour construction phase, it is possible to take benefit of the texture cache to reduce their access times.

Also, in order to compute the state transition rule, random numbers need to be generated and that feature is not directly available on GPUs. For that matter, a possible solution is to compute them prior to the beginning of the iterations and to store them in texture memory to enable faster access [33]. However, the great quantity of numbers needed by MMAS implies important CPU–GPU transfers. Therefore, the adopted solution is to implement the Linear Congruential Generator (LCG) procedure on GPU as it was proposed by Yu et al. [32] to initialize the seeds on the CPU and to let each ant use its own local numbers while constructing tours.

```
Initialize colony parameters, seed data structure and pheromone matrix $\tau$
Copy $\tau$, distance and candidate matrices in GPU texture memory
Copy colony data structure and seed on GPU global memory
Initialize number of blocks nBlocks and threads per block nThreads
for $t = 1$ to $ni$ do
    parallel region with nBlocks blocks of nThreads threads
        Retrieve id of ant: thread ($ANT_{thread}$) or block ($ANT_{block}$) identifier
        Initialize pseudorandom generator with seed of id
        TOUR CONSTRUCTION for $ant_{id}$
    parallel region with nBlocks blocks of nThreads threads
        Retrieve id of tour: thread or block identifier
        LOCAL SEARCH for $T_{id}$
    Retrieve constructed tours $T$ from GPU
    Update $T_{it}$, $T_{gb}$ and $\tau$
    Copy $\tau$ in GPU texture memory
```

Fig. 2. ANT_{thread} and ANT_{block} pseudo-code.

Fig. 2 describes a simplified pseudo-code of ANT_{thread} and ANT_{block} parallel strategies. Even if they are defined and evaluated in the context of MMAS, they are general enough to be adapted to most ACO implementations. Also, it is well known that current best performing ACO algorithms improve the solutions generated by the ants with local search algorithms, as it is the case with MMAS for the TSP [28].

4.3. Adding local search to the parallel ants GPU strategies

Integrating a standard 3-opt local search to MMAS implies that an important part of the global computation time will be devoted to it. Fig. 3(a) shows the general model of the execution of MMAS with local search. As each solution is being improved independently of the others, this step is a good candidate for parallelization. Not only is the local search well-suited for integration in the general parallel ants scheme, but the previously proposed strategies provide a natural framework for the GPU acceleration of this procedure. In both cases, the 3-opt procedure uses the distance matrix and candidates lists so it benefits from the texture cache of global memory.

Fig. 3(b) shows the 3-opt integration to the ANT_{thread} strategy. A thread is not associated only to an ant anymore but to the whole construction and improvement of a tour. In order to improve a current solution, local search generates different neighbors deleting arcs and reconnecting partial tours. These neighbors are then evaluated and the best one replaces the current solution. For the same reasons discussed in Section 4.2, the data structures necessary to compute the 3-opt process for all the ants of a given block can be stored only in global memory.

On the other hand, the ANT_{block} strategy provides a framework where further improvements may be proposed to the GPU 3-opt procedure. In fact, it renders the possibility of associating the whole construction and improvement of a tour to a block. The computation of the multiple neighbors is then shared between all its threads. This scheme is illustrated in Fig. 3(c). In the ANT_{block}^{global} strategy, the necessary data structures for the local search process are kept in global memory. However, the ANT_{block}^{shared} strategy opens the way for another improvement, which is to move all data to shared memory.

4.4. Multiple ant colonies

In the multiple ant colonies general parallelization approach, originally described by Stützle and Hoos [27], several ant colonies are distributed to processing elements. In that case, two possible hardware configurations are considered: single GPU and multiple GPU. This leads to two different strategies for independent ant colonies: $COLONY_{block}$ and $COLONY_{GPU}$. Moreover, since ants are executed on the GPU underlying architecture in both cases, they also take benefit of the strategies described in Sections 4.2 and 4.3. Fig. 4 illustrates the general model of this approach.

The $COLONY_{block}$ strategy associates each colony to a different block to be executed by a single SM of a GPU and applies the ANT_{thread} strategy to distribute the ants of a colony to threads of a block. Some data structures, like pheromone matrices, ants tours and various parameters, must be created for each colony. In order to limit prohibitive CPU–GPU transfers for this huge amount of data, the whole algorithm is executed on GPU. Distance and candidate matrices are not modified during the execution of the colonies so they may take advantage of the texture cache. On the other hand, pheromone matrices are subject to updates so they are kept in the global memory. Accordingly, all data structures are created on CPU for each colony and then copied in GPU memory. At the end of the algorithm where each block has completed all the iterations of its associated colony, the shortest tour of each colony is retrieved from GPU to CPU in order to determine the global best solution. Fig. 5 provides a pseudo-code of the $COLONY_{block}$ strategy.

The $COLONY_{GPU}$ strategy assumes a computing environment where many GPUs are interconnected and associates each colony to a different GPU. In that case, both parallel ants strategies of Section 4.2 may be applied and each ant of the colony may correspond to a thread or to a block of that GPU. This strategy implies some

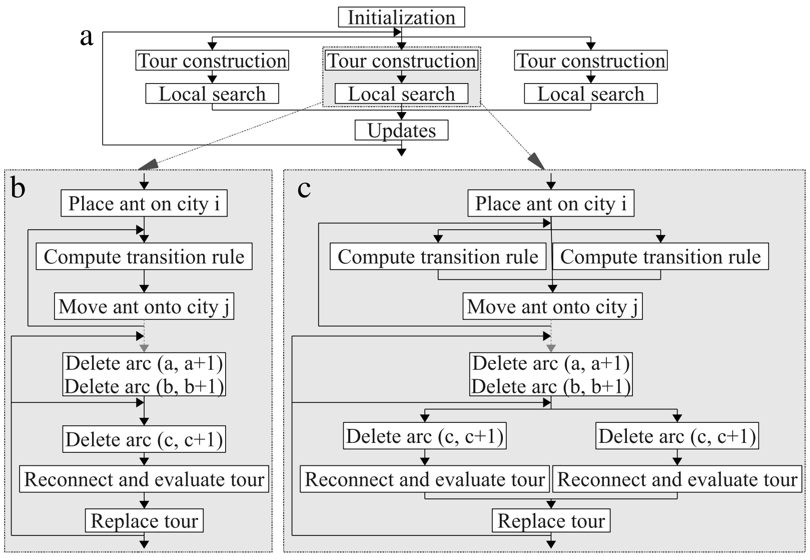

Fig. 3. Parallelization models of tour construction and local search for MMAS: general model (a), ANT_{thread} (b) and ANT_{block} (c).

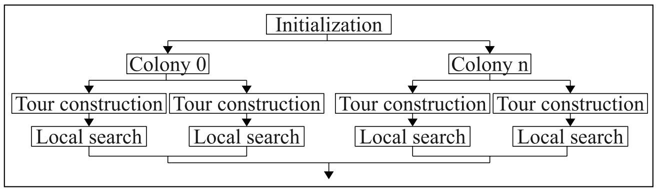

Fig. 4. Parallelization model of multi-colony MMAS.

```
Initialize colonies parameters, $seed$ data structures and pheromone matrices $\tau$
Copy distance and candidate matrices in GPU texture memory
Copy colonies data structures, $seed$ and $\tau$ on global memory
Initialize number of blocks $nBlocks$ and threads per block $nThreads$
parallel region with $nBlocks$ blocks of $nThreads$ threads
    Retrieve block identifier $idCol$ and thread identifier $idAnt$
    for $t = 1$ to $ni$ do
        TOUR CONSTRUCTION for $ant_{idAnt}$ and LOCAL SEARCH for $T_{idAnt}$
        Update $T_{it}^{idCol}$, $T_{gl}^{idCol}$ and pheromone matrix $\tau^{idCol}$
Retrieve tour $T_{gl}^{idCol}$ of each colony $idCol$ from GPU and find the best $T_{gl}$
```

Fig. 5. *COLONY*$_{block}$ pseudo-code.

```
parallel CPU region with $nbTh$ threads
    Initialize colonies parameters
    Retrieve $id$ of GPU: CPU thread identifier
    Apply $ANT_{thread}$ or $ANT_{block}$ strategy on $GPU_{id}$
Retrieve tour $T_{gl}^{id}$ of each colony $id$ and find the best $T_{gl}$
```

Fig. 6. *COLONY*$_{GPU}$ pseudo-code.

form of CPU parallelization to manage each colony before and after GPU execution: a parallel region creates multiple CPU execution threads (which could take the form of threads or processes at the application level) and duplicates the algorithm and data for each colony. Each GPU then becomes the SIMT co-processor of a given thread, and possibly of a processing element if such an environment is available at the CPU level. After each thread has initialized its colony and transferred the associated data to its GPU, it launches its colony and either the *ANT*$_{thread}$ or *ANT*$_{block}$ strategy is applied. After the termination condition is met for each colony, each CPU thread retrieves its shortest tour. Finally, at the end of CPU parallel region, the best solution is kept as the solution of the problem. A pseudo-code of this strategy is presented in Fig. 6.

An extensive experimental study, explained in the next section, has been performed to evaluate and compare the four strategies in a state-of-the-art GPU computing environment.

5. Experimental results

For each general parallelization approach, the two specific GPU strategies for MMAS designed in Section 4 are experimented and compared on various TSPs with sizes varying from 51 to 2103 cities. Conforming to the experimental principles adopted by Stüzle and Hoos [28], minimums and averages are computed from 25 trials for problems with less than 1000 cities and from 10 trials for larger instances. An effort is made to keep the algorithm and parameters as close as possible to the original MMAS. In some cases where we propose slightly different parameters to make the algorithm a better fit for GPUs, our choices are justified and results are provided to show the impact on solution quality.

Following the guidelines of Barr and Hickman [4] and Alba [1], the *relative speedup* metric is computed on *mean execution times* to evaluate the performance of the proposed implementations. For the *ANT*$_{thread}$, *ANT*$_{block}$ and *COLONY*$_{block}$ strategies involving only 1 GPU, speedups are calculated 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. In the case of *COLONY*$_{GPU}$, parallel time is obtained with as many CPU (or cores in the case of multi-core processors) and GPU as there are colonies, one CPU–GPU combination being linked to each colony.

Experiments were made on two GPUs of a NVIDIA Fermi C2050 server available at the Centre de Calcul Régional Champagne-Ardenne. Each GPU contains 14 SMs, 32 SPs per SM, 48 KB of shared memory per SM and a warp size of 32. The server also includes two 4-core Xeon E5640 CPUs running at 2.67 GHz and 24 GB of DDR3 memory. Application code was written in the "C for CUDA V3.1" programming environment.

For each problem, the number of threads and blocks used for the GPU resolution were empirically chosen according to a preliminary study based on previous work by the authors [14]. At this point, optimal general configurations can hardly be determined beforehand since they depend on many technical constraints linked to the GPU architecture and programming environment as well as on the algorithmic design of the metaheuristic. Overall, even though it is generally recommended to use a high number of threads in GPU applications [9], a compromise had to be found in the case of ACO algorithms. Consequently, thread and block configurations are provided for all experiments.

Next two sections provides results for parallel ants and multiple ant colonies approaches.

5.1. Parallel ants

The first part of the proposed experiments is to evaluate the performance of the *ANT*$_{thread}$ and *ANT*$_{block}$ strategies on a basic MMAS. Stützle and Hoos [28] recommend to use of a number of ants m equal to the number of cities n. However, CUDA documentation [9] strongly advise to use a number of threads per block that is a multiple of warp size to maximize efficiency. In order to take both considerations into account, we have chosen the values of m such as they are the multiple of warp size closest to n (in that case, $m = 32 \times 2^x$ where x is some integral number). In the *ANT*$_{thread}$ strategy, blocks and threads configurations always verify $m =$ *number of blocks x number of threads* with 64 blocks used for each problem. In the *ANT*$_{block}$ strategies, m blocks are used, each one of them being composed of a number of threads equal to the size of candidate lists *cl*, in that case 20. MMAS original authors also set the number of iterations to $2500n$ with $m = n$. Consequently, in our case it is set to $\frac{2500n}{m}$ with the intent of globally keeping the same global number of tour constructions.

A first step in our experiments is to compare solution quality obtained by sequential and parallel versions of the algorithm. Table 1 presents minimum and average tour lengths for each strategy and for each problem. The reader may first note the similarity between the results obtained by our sequential implementation and the ones provided by the authors of the original MMAS, as well as their closeness to optimal solutions. Results provided for all parallel strategies are also similar, showing that solution quality is globally preserved.

A second step is to evaluate and compare the reduction of execution time that is obtained with each parallelization strategy. Fig. 7 shows the speedups obtained for each problem. The reader may notice that the best speedup of *ANT*$_{thread}$ strategy is 5.84, which is greatly lower than the best ones of 15.50 and 19.47 obtained with the two variants of the *ANT*$_{block}$ strategy. Overall, *ANT*$_{thread}$ speedups are many times lower than *ANT*$_{block}$ ones for each problem and the gap becomes larger as problem size increases. This great difference comes mainly from code divergence induced by computing the state transition rule of many ants on the same block in SIMD mode, as well as from the limited amount of threads and blocks required to effectively hide memory latencies. Nevertheless, speedup generally increases with problem size, indicating that the strategy is scalable to some extent. The slight speedup decreases

Table 1
Minimum and average tour lengths of basic MMAS.

Problem	Optimum	Stützle and Hoos	Sequential algorithm	ANT_{thread}	ANT_{block}^{global}	ANT_{block}^{shared}
	426		426	426	426	426
eil51		427.80	427.32	427.76	427.20	427.20
	21,282		21,282	21,282	21,282	21,282
kroA100		21,336.90	21314.36	21321.20	21305.84	21317.32
	15,780		15,913	15,864	15,850	15,851
d198		15952.30	15973.84	15976.16	15970.52	15961.64
	42,029		42,107	42,172	42,102	42,147
lin318		42346.60	42341.72	42315.84	42336.20	42325.32
	8 806		8923	8905	8894	8899
rat783		–	9042.44	9014.72	9054.68	9002.32
	22,249		24,201	24,344	23,764	23,938
fl1577		–	24490.30	24541.90	24204.30	24287.80
	80,450		82,378	82,566	81,891	82,547
d2103		–	82754.30	82749.50	82704.80	82756.00

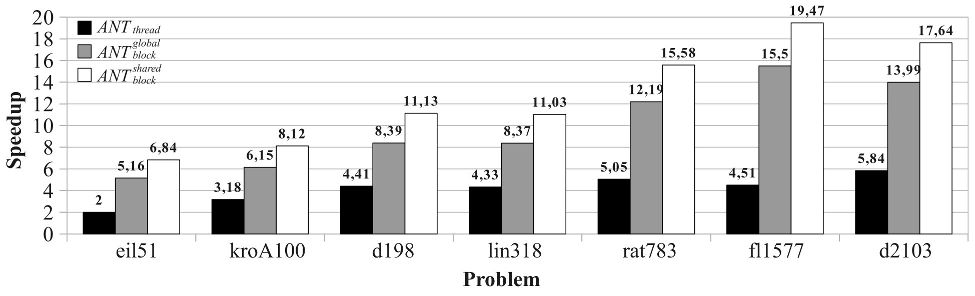

Fig. 7. Speedups of basic MMAS GPU implementations.

encountered with 318 and 1577 cities are due to structural differences that are specific to these problems.

The greater speedups obtained with the ANT_{block}^{global} strategy, ranging from 5.16 to 15.5, show that sharing the work of each ant between several threads is more efficient. Distributing the global work on a much higher number of threads is also beneficial for GPU execution. For example, for 2103 cities, it uses 40 960 threads versus 2048 for the ANT_{thread} strategy. This shows that the SIMT computation model of the GPU is better tailored to the computation of the state transition rule of single ants than to the execution of whole ants. Also, using different blocks favors ants independence during GPU execution.

Results also show that assigning ants to blocks brings further improvements by the use of shared memory. In fact, the ANT_{block}^{shared} strategy provides the best speedups for all problems, ranging from 6.84 to 19.47.

Overall, speedups of the ANT_{block} strategy increases more rapidly with problem size than the ANT_{thread} strategy, showing that this approach is more scalable. However, a slight decrease is encountered with the 2103 cities problem. In that case, the large workload and data structures imply memory access latencies and bank conflicts costs that grow faster than the benefits of parallelizing available work. Associated to the combined effect of the increasing number of blocks required to perform computations and a limited number of active blocks per SM, performance gains become less significative.

An important objective of this work is to propose parallel GPU strategies that provide competitive solution quality for TSP. Following this idea, next section is dedicated to performance evaluation of parallel ants strategies with MMAS augmented with local search.

5.2. Parallel ants with local search

The next step in our experiments is to evaluate the ANT_{thread} and ANT_{block} strategies when MMAS is augmented with a 3-opt local search. As the smallest problems are of little interest in this context, only the problems ranging from 198 to 2103 cities are tested. As it was explained in Section 5.1, algorithm parameters are set to be as close as possible to Stützle and Hoos [28] while taking GPU constraints into account. The size of candidate lists used for local search is set to 40. Even though 25 ants are usually used for tour construction, we choose to use 28 as this value is a multiple of the number of available SM. Also, in original MMAS work, authors limit the algorithm execution to a fixed maximum time then calculate the average number of iterations needed to find the optimal solution. As this method introduces a bias in speedup evaluation and comparison, we choose to use a fixed number of iterations for each problem. This value is set to 2048 as it is the first power of 2 higher than the needed iterations for solving bigger problems by the sequential algorithm. As this value is also easily divisible by the usual numbers used in GPU configurations, this ensures that the same number of tour constructions is performed in all cases no matter how many blocks and threads are used.

The number of blocks and threads used for each problem is as follows. For the ANT_{thread} strategy, 28/1 (28 blocks of 1 thread) is used in both tour construction and local search phases. For the ANT_{block} strategy, 28/20 is used for tour construction tour phase and 28/192 (d198), 28/160 (lin318), 28/160 (rat783), 28/224 (fl1577) and 28/192 (d2103) are used for local search.

In the same way as Section 5.1, Table 2 provides minimum and average tour lengths for each strategy. The reader may note that overall results are similar to those presented by Stützle and Hoos [28]. They are also close to optima. Moreover, results of parallel strategies are similar to those of sequential strategy, indicating that solution quality is globally preserved.

Fig. 8 shows the speedups obtained with each strategy. The reader may first note that speedups as high as 8.03 are obtained with this version of MMAS. Results also show that the ANT_{block} strategy leads to the best speedups in that case. However, when compared to Fig. 7, they indicate that speedups are lower when local search is applied to MMAS. This is explained in two different ways. On one hand, the structure of 3-opt local search is not very well suited to the GPU architecture. In fact, it requires few

Table 2
Minimum and average tour lengths of MMAS with local search.

Problem	Optimum	Stützle and Hoos	Sequential algorithm	ANT_{thread}	ANT_{block}^{global}	ANT_{block}^{shared}
d198	15,780	15,780	15,780	15,780	15,780	15,780
		15780.20	15780.08	15780.08	15780.04	15780.04
lin318	42,029	42,029	42,029	42,029	42,029	42,029
		42061.70	42036.44	42033.96	42041.40	42036.04
rat783	8 806	8806	8806	8808	8807	8806
		8816.80	8826.04	8824.76	8828.52	8828.68
fl1577	22,249	22,261	22,257	22,264	22,262	22,262
		22271.80	22273.50	22274.50	22279.30	22286.10
d2103	80,450	–	80,495	80,533	80,537	80,522
		–	80585.20	80563.00	80648.30	80682.40

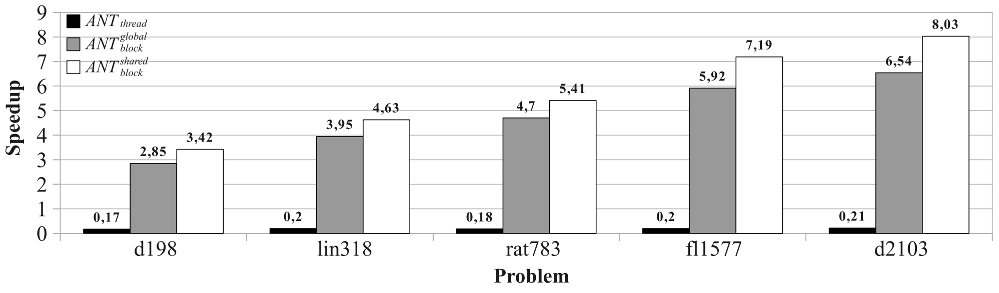

Fig. 8. Speedups of MMAS with local search GPU implementations.

Table 3
Blocks/threads configurations used for tour construction and local search for 2048 iterations and 28 ants (A), 256 iterations and 224 ants (B) and 32 iterations and 1792 ants (C) for 198 and 2103 cities.

Problem		ANT_{thread}		ANT_{block}^{global}		ANT_{block}^{shared}	
d198	A	28/1	28/1	28/20	28/192	28/20	28/192
	B	224/1	112/2	224/20	224/96	224/20	224/96
	C	28/64	112/16	1792/20	1792/96	1792/20	1792/64
d2103	A	28/1	28/1	28/20	28/192	28/20	28/192
	B	112/2	112/2	224/20	224/128	224/20	224/192
	C	56/32	112/16	1792/20	1792/128	1792/20	1792/192

Table 4
Speedups (boldface) and average tour lengths for 2048 iterations and 28 ants (A), 256 iterations and 224 ants (B) and 32 iterations and 1792 ants (C) for 198 and 2103 cities.

Problem		ANT_{thread}		ANT_{block}^{global}		ANT_{block}^{shared}	
d198	A	**0.17**	15780.08	**2.85**	15780.04	**3.42**	15780.04
	B	**0.62**	15780.32	**9.61**	15780.36	**11.28**	15780.28
	C	**0.78**	15780.00	**10.48**	15780.40	**12.48**	15780.52
d2103	A	**0.21**	80563.00	**6.52**	80641.10	**8.03**	80682.40
	B	**0.74**	80672.83	**10.01**	80938.30	**8.25**	81034.00
	C	**0.82**	82352.83	**10.02**	82775.30	**8.15**	82638.90

calculations compared to the high amount of read and write accesses to GPU memory needed to generate and evaluate the neighborhood of a current solution. On the other hand, the low number of ants implies that only 28 blocks or 28 threads are executed at each iteration, which is not enough work to efficiently exploit GPU resources and hide memory latency. The ANT_{thread} strategy provides the worst case as indicated by the absence of speedup. Speedups obtained by the ANT_{block}^{global} strategy, ranging from 2.85 to 6.54, show that this division of the work and the use of a higher number of threads is more efficient. Further improvements are brought by the use of shared memory of the ANT_{block}^{shared} strategy, which provides speedups between 3.42 and 8.03. On a last note, speedups that increase with problem size show the scalability of the ANT_{block} strategy on these problems, even with the local search limitations.

In order to provide some insight on the influence of the amount of work during tour construction and local search, we have tested increasing the number of ants and decreasing the number of iterations. Speedup and solution quality are thus always evaluated with the same global number of tour constructions. Table 3 shows the block/thread configurations used for tour construction and local search with each of the three iteration/ant combinations tested for 198 and 2103 cities problems. Table 4 respectively show sequential times, speedups and average tour lengths.

The reader may note that raising the number of ants and lowering the number of iterations generally has a positive effect on speedup for all strategies. This effect is negligible for the ANT_{thread} strategy speedups as they are still lower than 1. Not only is the number of threads still too low, but having many ants on the same block that apply the state transition rule and local search differently induces a great deal of thread divergence and serialization. The speedup increase is much more noticeable for the ANT_{block} strategy, going up to 12.48 for the 198 cities problem. In this case, the ANT_{block}^{shared} strategy also performs better than ANT_{block}^{global} strategy, which highlights the benefits of using shared memory. However, for the 2103 cities problem, speedup keeps increasing for ANT_{block}^{global} whereas it remains similar for ANT_{block}^{shared}. Speedup values are also higher in the first case. Since shared memory is limited in size, using too much of it reduces the number of active blocks per SM. Thus, for this specific case, experiments showed that only 2 blocks were active compared to 6 for ANT_{block}^{global} strategy. Moreover, for bigger problems, shared memory is not even large enough to store necessary data structures for a single block, indicating that the limits of shared memory are reached.

Concerning solution quality, Table 4 shows that raising the number of ants and lowering the number of iterations has no effect on average tour length for the 198 cities problem, whereas it has a negative effect for the 2103 cities problem. This shows that changing the dynamics of MMAS may lead to improvements in execution times, but at the expense of solution quality. Even though it might be possible to find specific parameters that favor speedup without hindering solution quality, the objective of this work was to relate to the original MMAS so it was not experimented extensively.

The second step of our experiments is related to multi-colony parallelization of MMAS. Results are presented in next section.

Table 5
$COLONY_{block}$ average tour lengths and speedups with different numbers of colonies (c.) and iterations (i.)

	1 c. 256 i.	2 c. 128 i.	4 c. 64 i.	10 c. 25 i.	256 c. 1 i.
Length average	15780.16	15780.17	15780.08	15789.00	15844.20
Speedup	0.06	0.15	0.30	0.73	1.77

Table 6
$COLONY_{GPU}$ speedups (boldface) and average tour lengths with different numbers of colonies and iterations.

Problem	1 colony, 256 iterations		2 colonies, 128 iterations	
d198	**9.61**	15780.36	**16.24**	15780.16
lin318	**12.70**	42071.52	**23.60**	42050.52
rat783	**9.38**	8830.96	**18.52**	8973.36
fl1577	**9.83**	22378.20	**19.66**	22461.30
d2103	**10.01**	80938.30	**19.74**	81223.20

5.3. Multiple ant colonies

Parallelization strategies proposed in Section 4.4 were tested on problems from 198 to 2103 cities. As a starting point for comparisons, the number of iterations is set to 256 and the number of ants per colony is set to 224 for sequential CPU execution as it was the case in Section 5.2. This ensures that there will be enough ants in each colony to make the search significative and provides an acceptable compromise between speedup and solution quality for all problems. Experiments are based on keeping the same global number of tour constructions for each configuration. Therefore, when the number of colonies is increased, the number of iterations is decreased.

For the $COLONY_{block}$ strategy, the number of blocks and threads are set to the number of colonies and to the number of ants per colony respectively. Average tour lengths and speedups for this strategy with 1, 2, 4, 10 and 256 colonies are shown in Table 5 for the 198 cities problem.

The reader may notice that speedup is achieved only with 256 colonies and that this comes at the expense of solution quality. Since all the ants of a colony are associated to a single thread of its block, these threads have to be synchronized during tour construction and local search. Moreover, pheromone matrices associated to each colony are updated so they do not take advantage of texture cache anymore. Deporting the entire algorithm on GPU also involves a great number of registers which dramatically lowers the number of active blocks per SM. Therefore, memory latency is not efficiently hidden. Overall, this strategy seems to offer limited potential and is not investigated more deeply.

In order to apply the $COLONY_{GPU}$ strategy to the studied problems, the two available GPUs are used. The ANT_{block}^{global} strategy is integrated within each colony as it offers the best performance without presenting shared memory limitations on bigger problems. 224 blocks of 20 threads are used within each GPU for the tour construction phase. Blocks/threads configurations used for local search phase are 224/96 for d198, lin318, rat783 and fl1577, and 224/128 for d2103. Table 6 presents average tour lengths and speedups obtained for 1 colony of 224 ants performing 256 iterations in comparison to 2 colonies of 224 ants performing 128 iterations on each GPU.

Results show that speedups range from 16.24 to 23.60 when 2 colonies are used. They are also approximately doubled in all cases when compared to a single colony. This shows that a multiple GPU parallelization is efficient. However, performing a lower number of iterations aggravates solution quality for the biggest problems. In this context, a way to improve tour lengths is to add information exchange strategies between colonies. We plan to address this issue in future work.

6. Conclusion

The aim of this paper was to propose efficient parallelization strategies for the implementation of Ant Colony Optimization on Graphics Processing Units. Following the *parallel ants* general approach, the ANT_{thread} and ANT_{block} strategies aimed at associating the ants tour construction and local search phases to the execution of streaming processors and multiprocessors respectively. On the other hand, the $COLONY_{block}$ and $COLONY_{GPU}$ strategies implemented the *multiple ant colonies* approach by attributing entire ant colonies to multiprocessors and whole GPUs respectively. We showed that both general approaches can be efficiently implemented on a GPU architecture. In fact, the ANT_{block} strategy managed to provide speedups as high as 19.47 with the basic MMAS and 12.48 with the addition of a 3-opt local search procedure. Speedups raise even higher with the $COLONY_{GPU}$ strategy, reaching 23.60 with the combination of MMAS and local search. Overall, this shows that it is possible to significantly reduce the execution time of ACO on GPU while rigorously keeping the similar competitive solution quality of the sequential MMAS.

Still, as it is the case in the field of parallel ACO and parallel metaheuristics in general, much can still be done for the effective use of GPUs. In fact, the variety of the proposed strategies and the extensive comparative study provided in this paper brings its share of questions and research avenues. For example, even though the use of the GPU shared memory leads to some of the best speedups, this hardware feature also shows its limits on bigger TSPs. Moreover, maximal exploitation of GPU resources often requires algorithmic configurations that do not let ACO perform an effective exploration and exploitation of the search space. Globally, this paper shows that parallel performance is strongly influenced by the combined effects of parameters related to the metaheuristic, the GPU technical architecture and the granularity of the parallelization. As it becomes clear that the future of computers no longer relies on increasing the performance on a single computing core but on using many of them in a single system, it becomes desirable to adapt optimization tools for parallel execution on architectures like GPUs.

Following this line of thought, our future works are aimed at using the framework and knowledge built in this paper to propose an ACO metaheuristic that is specifically tailored to GPU execution. Also, in order to provide better insight into the memory and algorithmic bottlenecks that have been identified in this work, a more formal analysis would be likely required. For example, the requirements for the different types of memory (main, device, shared, etc.) could be analyzed as a function of problem size, numbers of ants, colonies, SM, SP, etc. Such an analysis could lead to the proposition of algorithms that automatically determine effective thread/block/GPU configurations for ACO 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.

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] E. Alba, Parallel evolutionary algorithms can achieve super-linear performance, Information Processing Letters 82 (2002) 7–13.
[2] E. Alba, G. Leguizamon, G. Ordonez, Two models of parallel aco algorithms for the minimum tardy task problem, International Journal of High Performance Systems Architecture 1 (1) (2007) 50–59.
[3] H. Bai, D. Yang, X. Li, L. He, H. Yu, Max–min ant system on gpu with cuda, in: Fourth International Conference on Innovative Computing, Information and Control, IEEE, 2009, pp. 801–804.
[4] R.S. Barr, B.L. Hickman, Reporting computational experiments with parallel algorithms : issues, measures and experts' opinions, ORSA Journal on Computing 5 (1) (1993) 2–18.
[5] B. Bullnheimer, G. Kotsis, C. Strauss, Parallelization strategies for the ant system, in: R. De Leone, A. Murli, P. Pardalos, G. Toraldo (Eds.), High Performance Algorithms and Software in Nonlinear Optimization, in: Applied Optimization, vol. 24, Kluwer, Dordrecht, 1997, pp. 87–100.
[6] A. Catala, J. Jaen, J. Mocholi, Strategies for accelerating ant colony optimization algorithms on graphical processing units, in: IEEE Congress on Evolutionary Computation, IEEE Press, 2007, pp. 492–500.
[7] D. Chu, M. Till, A. Zomaya, Parallel ant colony optimization for 3d protein structure prediction using the hp lattice model, in: 19th IEEE International Parallel and Distributed Processing Symposium, vol. 7, IEEE Computer Society, 2005.
[8] M. Craus, L. Rudeanu, Parallel framework for ant-like algorithms, in: Third International Symposium on Parallel and Distributed Computing, ISPDC/HeteroPar'04, 2004, pp. 36–41.
[9] CUDA : Computer Unified Device Architecture Programming Guide 3.1 (2010). http://www.nvidia.com.
[10] P. Delisle, M. Gravel, M. Krajecki, Multi-colony parallel ant colony optimization on smp and multi-core computers, in: Proceedings of the World Congress on Nature and Biologically Inspired Computing, NaBIC 2009, IEEE, 2009, pp. 318–323.
[11] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, W.L. Price, A shared memory parallel implementation of ant colony optimization, in: 6th Metaheuristics International Conference, MIC'2005, Vienna, Autria, 2005, pp. 257–264.
[12] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, W.L. Price, Comparing parallelization of an aco: message passing vs. shared-memory, in: M.J. Blesa, C. Blum, A. Roli, M. Sampels (Eds.), in: Lecture Notes in Computer Science, vol. 3636, Springer-Verlag, Berlin Heidelberg, 2005, pp. 1–11.
[13] P. Delisle, M. Krajecki, M. Gravel, C. Gagné, Parallel implementation of an ant colony optimization metaheuristic with openmp, in: International Conference on Parallel Architectures and Compilation Techniques, 3rd European Workshop on OpenMP, EWOMP'01, Barcelona, Spain, 2001.
[14] A. Delévacq, P. Delisle, M. Gravel, M. Krajecki, Parallel ant colony optimization on graphics processing units, in: H.R. Arabnia, S.C. Chiu, G.A. Gravvanis, M. Ito, K. Joe, H. Nishikawa, A.M.G. Solo (Eds.), Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'10, CSREA Press, 2010, pp. 196–202.
[15] K. Doerner, R. Hartl, S. Benker, M. Lucka, Parallel cooperative savings based ant colony optimization—multiple search and decomposition approaches, Parallel Processing Letters 16 (3) (2006) 351–370.
[16] M. Dorigo, L.M. Gambardella, Ant colonies for the traveling salesman problem, BioSystems 43 (1997) 73–81.
[17] M. Dorigo, T. Stützle, Ant Colony Optimization, MIT Press, Bradford Books, 2004.
[18] I. Ellabib, P. Calamai, O. Basir, Exchange strategies for multiple ant colony system, Information Sciences 177 (5) (2007) 1248–1264.
[19] M.T. Islam, P. Thulasiraman, R.K. Thulasiram, A parallel ant colony optimization algorithm for all-pair routing in manets, in: 17th international Symposium on Parallel and Distributed Processing, IEEE Computer Society, 2003.
[20] J. Li, X. Hu, Z. Pang, K. Qian, A parallel ant colony optimization algorithm based on fine-grained model with gpu-acceleration, International Journal of Innovative Computing, Information and Control 5 (11(A)) (2009) 3707–3716.
[21] S. Lin, Computer solutions for the traveling salesman problem, Bell Systems Technical Journal 44 (1965) 2245–2269.
[22] M. Manfrin, M. Birattari, T. Stützle, M. Dorigo, Parallel ant colony optimization for the traveling salesman problem, in: Lecture Notes in Computer Science, vol. 4150, 2006, pp. 224–234.
[23] M. Middendorf, F. Reischle, H. Schmeck, Multi colony ant algorithms, Journal of Heuristics 8 (3) (2002) 305–320.
[24] M. Randall, A. Lewis, A parallel implementation of ant colony optimization, Journal of Parallel and Distributed Computing 62 (2) (2002) 1421–1432.
[25] B. Scheuermann, S. Janson, M. Middendorf, Hardware-oriented ant colony optimization, Journal of Systems Architecture 53 (2007) 386–402.
[26] B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. ElGindy, H. Schmeck, Fpga implementation of population-based ant colony optimization, Applied Soft Computing 4 (2004) 303–322.
[27] T. Stützle, Parallelisation strategies for ant colony optimization, in: A. Eiben, T. Bäck, H.-P. Schwefel, M. Schoenauer (Eds.), Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature, PPSN V, Springer-Verlag, New York, 1998.
[28] T. Stützle, H. Hoos, Max–min ant system, Future Generation Computer Systems 16 (8) (2000) 889–914.
[29] E. Talbi, O. Roux, C. Fonlupt, D. Robillard, Parallel ant colonies for the quadratic assignment problem, Future Generation Computer Systems 17 (4) (2001) 441–449.
[30] J. Wang, J. Dong, C. Zhang, Implementation of ant colony algorithm based on gpu, in: E. Banissi, M. Sarfraz, J. Zhang, A. Ursyn, W.C. Jeng, M.W. Bannatyne, J.J. Zhang, L.H. San, M.L. Huang (Eds.), Sixth International Conference on Computer Graphics, Imaging and Visualization: New Advances and Trends, IEEE Computer Society, 2009, pp. 50–53.
[31] Y. You, Parallel ant system for traveling salesman problem on gpus, in: GECCO 2009—Genetic and Evolutionary Computation, 2009, pp. 1–2.
[32] Q. Yu, C.C. amd, Z. Pan, Parallel genetic algorithms on programmable graphics hardware, in: Lecture Notes in Computer Science—Advances in Natural Computation, vol. 3612, Springer, Berlin, Heidelberg, 2005, pp. 1051–1059.
[33] W. Zhu, J. Curry, Parallel ant colony for nonlinear function optimization with graphics hardware acceleration, in: Proceedings of the 2009 IEEE international conference on Systems, Man and Cybernetics, IEEE Press, 2009, pp. 1803–1808.

Audrey Delévacq received her Master's degree in Computer Science from the École Pratique des Hautes Études – Paris, France, in 2009. She is currently completing her Ph.D Thesis on parallel GPU metaheuristics in the Centre de Recherche en Sciences et Technologies de l'Information et de la Communication (CReSTIC) laboratory of the Université de Reims Champagne-Ardenne, France. Her research interests are related to parallel metaheuristics and GPU computing.

Pierre Delisle received his Master's degree in Computer Science from the Université du Québec à Montréal, Canada, in 2002 and his Ph.D. degree in Computer Science from the Université de Reims Champagne-Ardenne, France, in 2006. He is an associate professor in the Centre de Recherche en Sciences et Technologies de l'Information et de la Communication (CReSTIC) laboratory of the Université de Reims Champagne-Ardenne since 2008. His research interests are in the fields of parallel computing, combinatorial optimization, metaheuristics and parallel implementation of optimization algorithms.

Marc Gravel is a full professor in the Université du Québec à Chicoutimi, Canada. He obtained a B.Sc degree in mathematics, an MBA degree from the Université Laval and Ph.D degree from the Université Aix-Marseilles III, France, in 1987. He is the author and co-author of numerous publications and directed the research work of several students. His research interests are related to production planning and control. In this regard, he has developed research collaborations with Canadian and French corporations in industrial scheduling.

Michaël Krajecki received his Master's degree in Computer Science from the Université de Metz, France, in 1995. He defended his Ph.D degree in Computer Science at the Université de Metz in 1998. He is a full professor in Computer Science from the Université de Reims Champagne-Ardenne since 2005. He is the actual head of the Centre de Recherche en Sciences et Technologies de l'Information et de la Communication (CReSTIC). He is also an associate professor in the Royal Military College of Canada, Ontario. His research interests are mainly focused on parallel algorithms, combinatorial optimization and high performance computing.