Chapter 3

Strategies for Parallel Ant Colony Optimization on Graphics Processing Units

Jaqueline S. Angelo, Douglas A. Augusto and Helio J. C. Barbosa

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/51679

1. Introduction

Ant colony optimization (ACO) is a population-based metaheuristic inspired by the collective behavior of ants which is used for solving optimization problems in general and, in particular, those that can be reduced to finding good paths through graphs. In ACO a set of agents (artificial ants) cooperate in trying to find good solutions to the problem at hand [1].

Ant colony algorithms are known to have a significant ability of finding high-quality solutions in a reasonable time [2]. However, the computational time of these methods is seriously compromised when the current instance of the problem has a high dimension and/or is hard to solve. In this line, a significant amount of research has been done in order to reduce computation time and improve the solution quality of ACO algorithms by using parallel computing. Due to the independence of the artificial ants, which are guided by an indirect communication via their environment (pheromone trail and heuristic information), ACO algorithms are naturally suitable for parallel implementation.

Parallel computing has become attractive during the last decade as an instrument to improve the efficiency of population-based methods. One can highlight different reasons to parallelize an algorithm: to (i) reduce the execution time, (ii) enable to increase the size of the problem, (iii) expand the class of problems computationally treatable, and so on. In the literature one can find many possibilities on how to explore parallelism, and the final performance strongly depends on both the problem they are applied to and the hardware available [3].

In the last years, several works were devoted to the implementation of parallel ACO algorithms [4]. Most of these use clusters of PCs, where the workload is distributed to multiple computers [5]. More recently, the emergence of parallel architectures such as multi-core processors and graphics processing units (GPU) allowed new implementations of parallel ACO algorithms in order to speedup the computational performance.

GPU devices have been traditionally used for graphics processing, which requires a high computational power to process a large number of pixels in a short time-frame. The massively parallel architecture of the GPUs makes them more efficient than general-purpose CPUs when large amount of independent data need to be processed in parallel.

The main type of parallelism in ACO algorithms is the parallel ant approach, which is the parallelism at the level of individual ants. Other steps of the ACO algorithms are also considered for speeding up their performance, such as the tour construction, evaluation of the solution and the pheromone update procedure.

The purpose of this chapter is to present a survey of the recent developments for parallel ant colony algorithms on GPU devices, highlighting and detailing parallelism strategies for each step of an ACO algorithm.

1.1. Ant Colony Optimization

Ant Colony Optimization is a metaheuristic inspired by the observation of real ants' behavior, applied with great success to a large number of difficult optimization problems.

Ant colonies, and other insects that live in colony, present interesting characteristics by the view of the collective behavior of those entities. Some characteristics of social groups in swarm intelligence are widely discussed in [6]. Among them, ant colonies in particular present a highly structured social organization, making them capable of self-organizing, without a centralized controller, in order to accomplish complex tasks for the survival of the entire colony [2]. Those capabilities, such as division of labor, foraging behavior, brood sorting and cooperative transportation, inspired different kinds of ant colony algorithms. The first ACO algorithm was inspired on the capability of ants to find the shortest path between a food source and their nest.

In all those examples ants coordinate their activities via *stigmergy* [7], which is an indirect communication mediated by modifications on the environment. While moving, ants deposit pheromone (chemical substance) on the ground to mark paths that may be followed by other members of the colony, which then reinforce the pheromone on that path. This behavior leads to a self-reinforcing process that results in path marked by high concentration of pheromone while less used paths tend to have a decreasing pheromone level due to evaporation. However, real ants can choose a path that has not the highest concentration of pheromone, so that new sources of food and/or shorter paths can be found.

1.2. Combinatorial problems

In combinatorial optimization problems one wants to find discrete values for solution variables that lead to the optimal solution with respect to a given objective function. An interesting characteristic of combinatorial problems is that they are easy to understand but very difficult to be solved [2].

One of the most extensively studied combinatorial problem is the Traveling Salesman Problem (TSP) [8] and it was the first problem approached by the ACO metaheuristic. The first developed ACO algorithm, called Ant System [1, 9], was initially applied to the TSP, then later improved and applied to many kinds of optimization problems [10].

http://dx.doi.org/10.5772/51679

In the Traveling Salesman Problem (TSP), a salesman, starting from an initial city, wants to travel the shortest path to serve its customers in the neighboring towns, eventually returning to the city where he originally came from, visiting each city once. The representation of the TSP can be done through a fully connected graph $G = (N, A)$, with N being the set of nodes representing cities and A the set of edges fully connecting the nodes. For each arc (i, j) is assigned a value d_{ij}, which may be distance, time, price, or other factor of interest associated with edges $a_{i,j} \in A$. The TSP can be symmetric or asymmetric. Using distances (associated with each arc) as cost values, in the symmetric TSP the distance between cities i and j is the same as between j and i, i.e. $d_{ij} = d_{ji}$; in the asymmetric TSP the direction used for crossing an arc is taken into consideration and so there is at least one arc in which $d_{ij} \neq d_{ji}$. The objective of the problem is to find the minimum Hamiltonian cycle, where a Hamiltonian cycle is a closed tour visiting each of the $n = |N|$ nodes (cities) of G exactly once.

2. Graphics Processing Unit

Until recently the only viable choice as a platform for parallel programming was the conventional CPU processor, be it single- or multi-core. Usually many of them were arranged either tightly as multiprocessors, sharing a single memory space, or loosely as multicomputers, with the communication among them done indirectly due to the isolated memory spaces.

The parallelism provided by the CPU is reasonably efficient and still very attractive, particularly for tasks with low degree of parallelism, but a new trendy platform for parallel computing has emerged in the past few years, the *graphics processing unit*, or simply the GPU architecture.

The beginning of the GPU architecture dates back to a couple of decades ago when some primitive devices were developed to offload certain basic graphics operations from the CPU. Graphics operations, which end up being essentially the task to determine the right color of each individual pixel per frame, are in general both independent and specialized, allowing a high degree of parallelism to be explored. However, doing such operations on conventional CPU processors, which are general-purpose and back then were exclusively sequential, is slow and inefficient. The advantage of parallel devices designed for such particular purpose was then becoming progressively evident, enabling and inviting a new world of graphics applications.

One of those applications was the computer game, which played an important role on the entire development history of the GPU. As with other graphics applications, games involve computing and displaying—possibly in parallel—numerous pixels at a time. But differently from other graphics applications, computer games were always popular among all range of computer users, and thus very attractive from a business perspective. Better and visually appealing games sell more, but they require more computational power. This demand, as a consequence, has been pushing forward the GPU development since the early days, which in turn has been enabling the creation of more and more complex games.

Of course, in the meantime the CPU development had also been advancing, with the processors becoming progressively more complex, particularly due to the addition of cache memory hierarchies and many specific-purpose control units (such as branch prediction, speculative and out-of-order execution, and so on) [11]. Another source of development has

been the technological advance in the manufacturing process, which has been allowing the manufactures to systematically increase the transistor density on a microchip. However, all those progresses recently begun to decline with the Moore's Law [12] being threatened by the approaching of the physical limits of the technology on the transistor density and operating frequency. The response from the industry to continually raise the computational power was to migrate from the sequential single-core to the parallel multi-core design.

Although the nowadays multi-core CPU processors perform fairly well, the decades of accumulative architectural optimizations toward sequential tasks have led to big and complex CPU cores, hence restricting the amount of them that could be packed on a single processor—not more than a few cores. As a consequence, the current CPU design cannot take advantage of workloads having high degree of parallelism, in other words, it is inefficient for massive parallelism.

Contrary to the development philosophy of the CPU, because of the requirements of graphics operations the GPU took since its infancy the massive parallelism as a design goal. Filling the processor with numerous ALUs[1] means that there is not much die area left for anything else, such as cache memory and control units. The benefit of this design choice is two-fold: (i) it simplifies the architecture due to the uniformity; and (ii) since there is a high portion of transistors dedicated to actual computation (spread over many ALUs), the theoretical computational power is proportionally high. As one can expect, the GPU reaches its peak of efficiency when the device is fully occupied, that is, when there are enough parallel tasks to utilize each one of the thousands of ALUs, as commonly found on a modern GPU.

Besides being highly parallel, this feature alone would not be enough to establish the GPU architecture as a compelling platform for mainstream high-performance computation. In the early days, the graphics operations were mainly primitive and thus could be more easily and efficiently implemented in hardware through fixed, i.e. specialized, functional units. But again, such operations were becoming increasingly more complex, particularly in visually-rich computer games, that the GPU was forced to switch to a programmable architecture, where it was possible to execute not only strict graphics operations, but also arbitrary instructions. The union of an efficient massively parallel architecture with the general-purpose capability has created one of the most exciting processor, the modern GPU architecture, outstanding in performance with respect to power consumption, price and space occupied.

The following section will introduce the increasingly adopted open standard for heterogeneous programming, including of course the GPU, known as OpenCL.

2.1. Open Computing Language – OpenCL

An interesting fact about the CPU and GPU architectures is that while the CPU started as a general-purpose processor and got more and more parallelism through the multi-core design, the GPU did the opposite path, that is, started as a highly specialized parallel processor and was increasingly endowed with general-purpose capabilities as well. In other words, these architectures have been slowly converging into a common design, although each one still has—and probably will always have due to fundamental architectural differences—divergent strengths: the CPU is optimized for achieving low-latency in sequential tasks whereas the GPU is optimized for maximizing the throughput in highly parallel tasks [13].

[1] *Arithmetic and Logic Unit*, the most basic form of computational unit.

http://dx.doi.org/10.5772/51679

It is in this convergence that OpenCL is situated. In these days, most of the processors are, to some extent, both parallel and general purpose; therefore, it should be possible to come along with a uniform programming interface to target such different but fundamentally related architectures. This is the main idea behind OpenCL, a platform for uniform parallel programming of heterogeneous systems [14].

OpenCL is an open standard managed by a non-profit organization, the Khronos Group [14], that is architecture- and vendor-independent, so it is designed to work across multiple devices from different manufactures. The two main goals of OpenCL are *portability* and *efficiency*. Portability is achieved by the guarantee that every supported device conforms with a common set of functionality defined by the OpenCL specification [15].[2] As for efficiency, it is obtained through the flexible multi-device programming model and a rich set of relatively low-level instructions that allow the programmer to greatly optimize the parallel implementation (possibly targeting a specific architecture if so desirable) without loss of portability.[3]

2.1.1. Fundamental Concepts and Terminology

An OpenCL program comprises two distinct types of code: the *host*, which runs sequentially on the CPU, and the *kernel*, which runs in parallel on one or more devices, including CPUs and GPUs. The host code is responsible for managing the OpenCL devices and setting up/controlling the execution of kernels on them, whereas the actual parallel processing is programmed in the kernel code.

2.1.1.1. Host code

The tasks performed by the host portion usually involve: (1) discovering and enumeration of the available compute devices; (2) loading and compilation of the kernels' source code; (3) loading of domain-specific data, such as algorithm's parameters and problem's data; (4) setting up kernels' parameters; (5) launching and coordinating kernel executions; and finally (6) outputting the results. The host code can be written in the C/C++ programming language.[4]

2.1.1.2. Kernel code

Since it implements the parallel decomposition of a given problem—a *parallel strategy*—, the kernel is usually the most critical aspect of an OpenCL program and so care should be taken in its design.

The OpenCL kernel is similar to the concept of a procedure in a programming language, which takes a set of input arguments, performs computation on them, and writes back the result. The main difference is that an OpenCL kernel is a procedure that, when launched, actually multiple instances of them are spawned simultaneously, each one assigned to an individual execution unit of a parallel device.

---

[2] In fact, all the parallel strategies described in Section 4 can be readily applied on a CPU device (or any other OpenCL-supported device, such as DSPs and FPGAs) without modification.

[3] Of course, although OpenCL guarantees the functional portability, i.e. that the code will run on any other supported device, doing optimizations aimed at getting the most out of a specific device or architecture may lead to the loss of what is known as *performance portability*.

[4] C and C++ are the only officially supported languages by the OpenCL specification, but there exist many other third-party languages that could also be used.

An instance of a kernel is formally called a *work-item*. The total number of work-items is referred to as *global size*, and defines the level of decomposition of the problem: the larger the global size, the finer is the granularity, and is always preferred over a coarser granularity when targeting a GPU device in order to maximize its utilization—if that does not imply in a substantial raise of the communication overhead.

The mapping between a work-item and the problem's data is set up through the concept known as *N-dimensional domain range*, or just N-D domain, where N denotes a one-, two-, or three-dimensional domain. In practice, this is the mechanism that connects the work-items execution ("compute domain") with the problem's data ("data domain"). More specifically, the OpenCL runtime assigns to each work-item a unique identifier, a $global_{id}$, which in turn makes it possible to an individual work-item to operate on a subset of the problem's data by somehow indexing these elements through the identifier.

Figure 1 illustrates the concept of a mapping between the compute and data domains. Suppose one is interested in computing in parallel a certain operation over an array of four dimensions ($n = 4$), e.g. computing the square root of each element. A trivial strategy would be to dedicate a work-item per element, but let us assume one wants to limit the number of work-items to just two, that is, $global_{size} = 2$. This means that a single work-item will have to handle two data elements, thus the granularity $g = 2$. So, how could one connect the compute and data domains? There are different ways of doing that, but one way is to, from within the work-item, index the elements of the *input* and *output* by the expression $g \times t + global_{id}$, where $t \in \{0, 1\}$ is the time step (iteration).

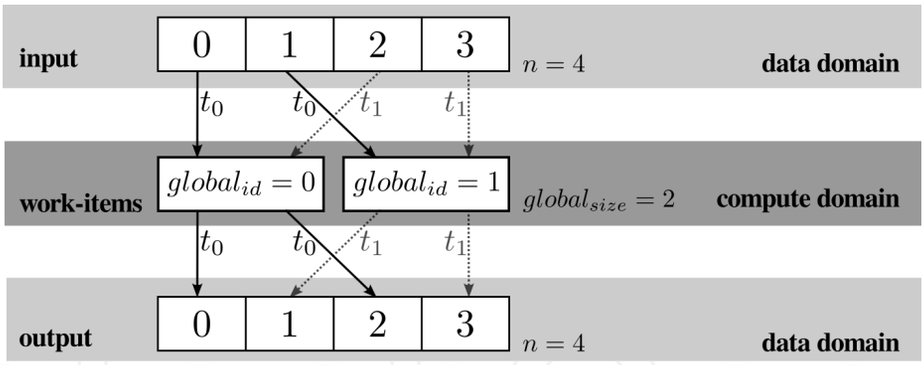

Figure 1. Example of a mapping between the compute and data domains.

A pseudo-OpenCL kernel implementing such strategy is presented in Algorithm 1.[5] At step t_0, the first and second work-items will be accessing, respectively, the indices 0 and 1, and at t_1 they will access the indices 2 and 3.

Algorithm 1: Example of a pseudo-OpenCL kernel

for $t \leftarrow 0$ **to** $\frac{n}{global_{size}} - 1$ **do**
  $\text{output}[g \times t + global_{id}] \leftarrow \sqrt{\text{input}[g \times t + global_{id}]}$;

[5] An actual OpenCL kernel is implemented in OpenCL C, which is almost indistinguishable from the C language, but adds a few extensions and also some restrictions [15].

http://dx.doi.org/10.5772/51679

The N-D domain range can also be extended to higher dimensions. For instance, in a 2-D domain a work-item would have two identifiers, $global_{id}^{0}$ and $global_{id}^{1}$, where the first could be mapped to index the *row* and the second the *column* of a matrix. The reasoning is analogous for a 3-D domain range.

2.1.1.3. Communication and Synchronization

There are situations in which it is desirable or required to allow work-items to *communicate* and *synchronize* among them. For efficiency reasons, such operations are not arbitrarily allowed among work-items across the whole N-D domain.[6] For that purpose, though, one can resort to the notion of *work-group*, which in a nutshell is just a collection of work-items. All the work-items within a work-group are free to communicate and synchronize with each other. The number of work-items per work-group is given by the parameter *local size*, which in practice determines how the global domain is partitioned. For example, if $global_{size}$ is 256 and $local_{size}$ is 64, then the computational domain is partitioned into 4 work-groups (256/64) with each work-group having 64 work-items. Again, the OpenCL runtime provides means that allow each work-group and work-item to identify themselves. A work-group is identified with respect to the global N-D domain through $group_{id}$, and a work-item is identified locally within its work-group via $local_{id}$.

2.1.2. Compute Device Abstraction

In order to provide a uniform programming interface, OpenCL abstracts the architecture of a parallel compute device, as shown in Figure 2. There are two fundamental concepts in this abstraction, the *compute* and *memory* hierarchies.

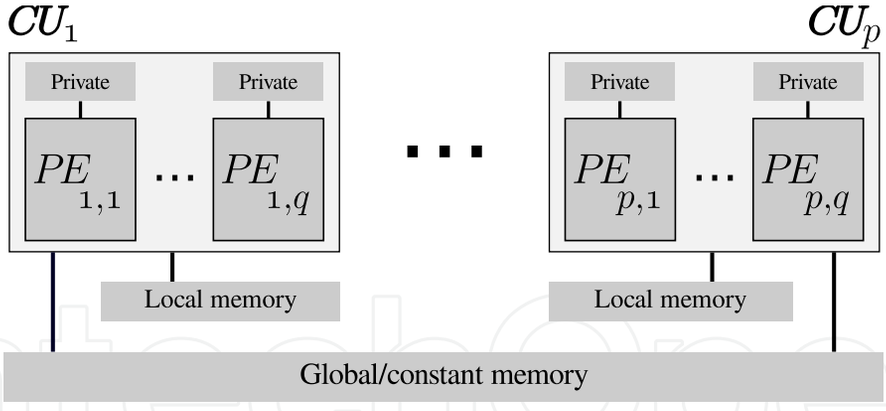

Figure 2. Abstraction of a parallel compute device architecture [16].

OpenCL defines two levels of compute hardware organization, the *compute units* (CU) and *processing elements* (PE). Not coincidentally this partitioning matches the software abstraction of work-groups and work-items. In fact, OpenCL guarantees that a work-group is entirely executed on a single compute unit whereas work-items are executed by processing elements. Nowadays GPUs usually have thousands of processing elements clustered in a dozen of

[6] There are two main reasons why those operations are restricted: (i) to encourage the better programming practice of avoiding dependence on communication as much as possible; and, most importantly, (ii) to allow the OpenCL to support even those rather limited devices that cannot keep—at least not efficiently—the state of all the running work-items as needed to fulfill the requirements to implement the global synchronization.

compute units. Therefore, to fully utilize such devices, there is needed at the very least this same amount of work-items in flight—however, the optimal amount of work-items in execution should be substantially more than that in order to the device have enough room to hide latencies [17, 18].

As for the memories, OpenCL exposes three memory spaces; from the more general to the more specific: the (i) *global/constant* memory, which is the main memory of the device, accessible from all the work-items—the *constant* space is a slightly optimized global memory for read-only access; (ii) the *local* memory, a very fast low-latency memory which is shared only across the work-items within their work-group—normally used as a programmable cache memory or as a means to share data (communicate); and (iii) the *private* memory, also a very fast memory, but only visible to the corresponding work-item.

3. Review of the literature

In the last few years, many works have been devoted to parallel implementations of ACO algorithms in GPU devices, motivated by the powerful massively parallel architecture provided by the GPU.

In reference [19], the authors proposed two parallel ACO implementations to solve the Orienteering Problem (OP). The strategies applied to the GPU were based on the intrinsically data-parallelism provided by the vertex processor and the fragment processor. The first experiments compared a grid implementation with 32 workstations equipped with CPUs Intel Pentium IV at 2.4GHz against one workstation with a GPU NVIDIA GeForce 6600 GT. Both strategies performed similarly with respect to the quality of the obtained solutions. The second experiment compared both the GPU parallel strategies proposed, showing that the strategy applied to the fragment processor performed about 35% faster than the strategy applied to the vertex processor.

In [20], the authors implemented a parallel $\mathcal{MM}$AS using multiple colonies, where each colony is associated with a work-group and ants are associated with work-items within each work-group. The experiments compared a parallel version of $\mathcal{MM}$AS on the GPU, with three serial CPU versions. In the parallel implementation the CPU initializes the pheromone trails, parameters, and also controls the iteration process, while the GPU is responsible for running the main steps of the algorithm: solution construction, choice of the best solution, and pheromone evaporation and updating. Six instances from the Travelling Salesman Problem library (TSPLIB), containing up to 400 cities, were solved using a workstation with a CPU AMD Athlon X2 3600+ running at 1.9GHz and a GPU NVIDIA GeForce GTX 8800 at 1.35GHz with 128 processing elements. The parallel GPU version was 2 to 32 times faster than the sequential version, whereas the solutions quality of the parallel version outperformed all the three $\mathcal{MM}$AS serial versions. In order to accelerate the choice of the iteration-best solution, the authors used a parallel reduction technique that "hangs up" the execution of certain work-items. This technique requires the use of barrier synchronization in order to ensure consistency of memory.

In the work described in [21] the authors implemented a parallel ACO algorithm with a pattern search procedure to solve continuous functions with bound constraints. The parallel method was compared with a serial CPU implementation. Each work-item is responsible for evaluating the solution's costs and constraints, constructing solutions and improving them

http://dx.doi.org/10.5772/51679

via a local search procedure, while the CPU controls the initialization process, pheromone evaporation and updating, the sorting of the generated solutions, and the updating of the probability vectors. The experiments were executed on a workstation equipped with a CPU Intel Xeon E5420 at 2.5GHz and a GPU NVIDIA GeForce GTX 280 at 1296MHz and 240 processing elements. The computational experiments showed acceleration values between 128 and almost 404 in the parallel GPU implementation. On the other hand, both the parallel and serial versions obtained satisfactory results. However, regarding the solution quality under a time limit of one second, the parallel version outperformed the sequential one in most of the test problems. As a side note, the results could have been ever better if the authors had generated the random numbers directly on the GPU instead of pre computing them on the CPU.

A parallel $\mathcal{MM}$AS under a MATLAB environment was presented in [22]. The authors proposed an algorithm implementation which arranges the data into large scale matrices, taking advantage of the fact that the integration of MATLAB with the Jacket accelerator handles matrices on the GPU more naturally and efficiently than it could do with other data types. Therefore, auxiliary matrices were created, besides the usual matrices (τ and η) in a standard ACO algorithm. Instances from the TSPLIB were solved using a workstation with a CPU Intel i7 at 3.3GHz and GPU NVIDIA Tesla C1060 at 1.3GHz and 240 processing elements. Given a fixed number of iterations, the experimental evaluation showed that the CPU and GPU implementations obtained similar results, yet with the parallel GPU version much faster than the CPU. The speedup values had been growing with the number of TSP nodes, but when the number of nodes reached 439 the growth could not be sustained and slowed down drastically due to the frequent data-transfer operations between the CPU and GPU.

In [23], the authors make use of the GPU parallel computing power to solve pathfinding in games. The ACO algorithm proposed was implemented on a GPU device, where the parallelism strategies follow a similar strategy to the one presented in [19]. In this strategy, ants works in parallel to obtain a solution to the problem. The author intended to study the algorithm scalability when large size problems are solved, against a corresponding implementation on a CPU. The hardware architecture was not available but the computational experiments showed that the GPU version was 15 times faster than its corresponding CPU implementation.

In [24] an ACO algorithm was proposed for epistasis[7] analysis. In order to tackle large scale problems, the authors proposed a multi-GPU parallel implementation consisting of one, three and six devices. The experiments show that the results generated by the GPU implementation outperformed two other sequential versions in almost all trials and, when the dataset increased, the GPU performed faster than the other implementations.

The Quadratic Assignment Problem (QAP) was solved in [25] by a parallel ACO based algorithm. Besides the initialization process, all the algorithm steps are performed on the GPU, and all data (pheromone matrix, set of solutions, etc.) are located in the global memory of the GPU. Therefore, no data was needed to be transferred between the CPU and GPU, only the best-so-far solution which checks if the termination condition is satisfied. The authors focus on a parallelism strategy for the 2-opt local search procedure since, from previews experiments, this was the most costly step. The experiments were done in a workstation

[7] Phenomenon where the effects of one gene are modified by one or several other genes.

with CPU Intel i7 965 at 3.2GHz and GPU NVIDIA GeForce GTX 480 at 1401MHz and 480 processing elements. Instances from the Quadratic Assignment Problem library (QAPLIB) were solved with the problem size ranging from 50 to 150. The GPU computing performed 24 times faster than the CPU.

An ACO based parallel algorithm was proposed for design validation of circuits [26]. The ACO method is different from the standard ACO implementation, since it does not use pheromones trails to guide the search process. The proposed method explores the maximum occupancy of the GPU, defining the global size as the number of work-groups times the amount of work-items per work-group. A workstation with CPU Intel i7 at 3.33GHz and a GPU NVIDIA GeForce GTX 285 with 240 processing elements were used for the computational experiments. The results showed average speedup values between 7 and 11 regarding all the test problems, and reaching a peak speedup value of 228 in a specific test problem when compared with two other methods.

In [27], the $\mathcal{MM}$AS with a 3-opt local search was implemented in parallel on the GPU. The authors proposed four parallel strategies, two based on parallel ants and two based on multiple ant colonies. In the first parallel-ants strategy, ants are assigned to work-items, each one responsible for all calculation needed in the tour construction process. The second parallel-ants proposal assigned each ant to a work-group, making possible to extract an additional level of parallelism in the computation of the state transition rule. In the multiple colony strategy, a single GPU and multiples GPUs—each one associated to a colony—were used, applying the same parallel-ants strategies proposed. TSP instances varying from 51 to 2103 cities were used as test problems. The experiments were done using two CPUs 4-core Xeon E5640 at 2.67GHz and two GPUs NVIDIA Fermi C2050 with 448 processing elements. Evaluating the parallel ants strategies against the sequential version of the $\mathcal{MM}$AS, the overall experiments showed that the solutions quality were similar, when no local search was used. However, speedup values ranging from 6.84 to 19.47 could be achieved when the ants were associated with work-groups. For the multiple colonies strategies the speedup varied between 16.24 and 23.60.

The authors in [28] proposed parallel strategies for the tour construction and the pheromone updating phases. In the tour construction phase three different aspects were reworked in order to increase parallelism: (i) the choice-info matrix calculation, which combines pheromone and heuristic information; (ii) the *roulette wheel* selection procedure; and (iii) the decomposition granularity, which switched to the parallel processing of both ants and tours. Regarding the pheromone trails updating, the authors applied a *scatter to gather* based design to avoid atomic instructions required for proper updating the pheromone matrix. The hardware used for the computational experiments were composed by a CPU Intel Xeon E5620 running at 2.4Ghz and a GPU NVIDIA Tesla C2050 at 1.15GHz and 448 processing elements. For the phase of the construction of the solution, the parallel version performed up to 21 times faster than the sequential version, while for the pheromone updating the scatter to gather technique performed poorly. However, considering a data-based parallelism with atomic instructions, the authors presented a strategy that was up to 20 times faster than a sequential execution.

The next section will present strategies for the parallel ACO on the GPU for each step of the algorithm.

http://dx.doi.org/10.5772/51679

4. Parallelization strategies

In ACO algorithms, artificial ants cooperate while exploring the search space, searching good solutions for the problem through a communication mediated by artificial pheromone trails. The construction solution process is incremental, where a solution is built by adding solution components to an initially empty solution under construction. The ant's heuristic rule probabilistically decides the next solution component guided by (i) the heuristic information (η), representing a priori information about the problem instance to be solved; and (ii) the pheromone trail (τ), which encodes a memory about the ant colony search process that is continuously updated by the ants.

The main steps of the Ant System (AS) algorithm [1, 9] can be described as: initialization phase, ants' solutions construction, ants' solutions evaluation and pheromone trails updating. In Algorithm 2 a pseudo-code of AS is given. As opposed to the following parallel strategies, this algorithm is meant to be implemented and run as host code, preparing and transferring data to/from the GPU, setting kernels' arguments and managing their executions.

Algorithm 2: Pseudo-code of Ant System.

```
// Initialization phase
Pheromone trails τ;
Heuristic information η;
// Iterative phase
while termination criteria not met do
    Ants' solutions construction;
    Ants' solutions evaluation;
    Pheromone trails updating;
Return the best solution;
```

After setting the parameters, the first step of the algorithm is the initialization procedure, which initializes the heuristic information and the pheromone trails. In ants' solution construction, each ant starts with a randomly chosen node (city) and incrementally builds solutions according to the decision policy of choosing an unvisited node j being at node i, which is guided by the pheromone trails (τ_{ij}) and the heuristic information (η_{ij}) associated with that arc. When all ants construct a complete path (feasible solution), the solutions are evaluated. Then, the pheromone trails are updated considering the quality of the candidate solutions found; also a certain level of evaporation is applied. When the iterative phase is complete, that is, when the termination criteria is met, the algorithm returns the best solution generated.

As showed in the previous section, different parallel techniques for ACO algorithms were proposed, each one adapted to the optimization problem considered and the GPU architecture available. In all cases, researchers tried to extract the maximum efficiency of the parallel computing provided by the GPU.

This section is dedicated to describe, in a pseudo-OpenCL form, parallelization strategies of the ACO algorithm described in Algorithm 2, taking the TSP as an illustrative reference

problem.[8] Those strategies, however, should be readily applicable, with minor or no adaptations at all, to all the problems that belong to the same class of the TSP.[9]

4.1. Data initialization

This phase is responsible for defining the stopping criteria, initializing the parameters and allocating all data structures of the algorithm. The list of parameters is: α and β, which regulate the relative importance of the pheromone trails and the heuristic information, respectively; ρ, the pheromone evaporation rate; τ_0, the initial pheromone value; number of ants ($number_{ants}$); and the number of nodes ($number_{nodes}$). The parameters setting is done on the host and then passed as kernel's arguments.

In the following kernels all the data structures, in particular the matrices, are actually allocated and accessed as linear arrays, since OpenCL does not provide abstraction for higher-dimensional data structures. Therefore, the element $a_{ij} \in A$ is indexed in its linear form as $A[i \times n + j]$, where n is the number of columns of matrix A.

4.1.1. Pheromone Trails and Heuristic Information

To initialize the pheromone trails, all connections (i, j) must be set to the same initial value (τ_0), whereas in the heuristic information each connection (i, j) is set as the distance between the nodes i and j of the TSP instance being solved. Since the initialization operation is inherently independent it can be trivially parallelized. Algorithm 3 presents the kernel implementation in which a 2-D domain range[10] is used and defined as

$$\begin{aligned} global^0_{size} &\leftarrow number_{nodes} \\ global^1_{size} &\leftarrow number_{nodes} \end{aligned} \tag{1}$$

Algorithm 3: OpenCL kernel for initializing τ and η

$\tau[global^0_{id} \times global^1_{size} + global^1_{id}] \leftarrow \tau_0$;
$\eta[global^0_{id} \times global^1_{size} + global^1_{id}] \leftarrow$ `Distance(`$x[global^0_{id}], y[global^1_{id}]$`);`

In the kernel, the helper function `Distance(`i, j`)` returns the distance between nodes i and j. The input data are two arrays with the coordinates x and y of each node. This function should implement the Euclidean, Manhattan or other distance function defined by the problem. The input coordinates must be set on the CPU, by reading the TSP instance, then transferred to the GPU prior to the kernel launch.

[8] In this chapter only the key components to the understanding of the parallel strategies—the OpenCL kernels and the corresponding setup of the N-dimensional domains—are presented. For specific details regarding secondary elements, such as the host code and the actual OpenCL kernel, please refer to the appropriated OpenCL literature.

[9] It might be necessary some adaptations concerning the algorithmic structure (data initialization, evaluation of costs, etc.) that might have particular needs with respect to the underlying strategy of parallelism.

[10] The OpenCL kernels presented throughout this chapter are either in a one- or two-dimensional domain range, depending on which one fits more naturally the particular mapping between the data and compute domains.

http://dx.doi.org/10.5772/51679

4.2. Solution construction

For the TSP, this phase is the most costly of the ACO algorithm and needs special attention regarding the parallel strategy.

In this section, a parallel implementation for the solution construction will be presented—the *ant-based* parallelism—which associates an ant with a work-item.

4.2.1. Caching the Pheromone and Heuristic Information

The probability of choosing a node j being at node i is associated with $[\tau_{ij}]^{\alpha}[\eta_{ij}]^{\beta}$. Each of those values need to be computed by all ants, hence, in order to reduce the computation time [2] an additional matrix, $\text{choice}_{info}[\cdot][\cdot]$, is utilized to cache them. For this caching computation, a 2-D domain range is employed and defined as

$$\begin{aligned} global^{0}_{size} &\leftarrow number_{nodes} \\ global^{1}_{size} &\leftarrow number_{nodes}, \end{aligned} \tag{2}$$

with the corresponding kernel described in Algorithm 4.

Algorithm 4: OpenCL kernel for calculating the *choice-info* cache

$$\begin{aligned} &\text{choice}_{info}[global^{0}_{id} \times global^{1}_{size} + global^{1}_{id}] \leftarrow \\ &\quad \tau[global^{0}_{id} \times global^{1}_{size} + global^{1}_{id}]^{\alpha} \times \eta[global^{0}_{id} \times global^{1}_{size} + global^{1}_{id}]^{\beta}; \end{aligned}$$

Whenever the pheromone trails τ is modified (4.1 and 4.4), the matrix choice_{info} also needs to be updated since it depends on the former. In other words, the caching data is recalculated at each iteration, just before the actual construction of the solution.

4.2.2. Ant-based Parallelism (AP)

In this strategy, each ant is associated with a work-item, each one responsible for constructing a complete solution, managing all data required for this phase (list of visited cities, probabilities calculations, and so on). Algorithm 5 presents a kernel which implements the AS decision rule, where the 1-D domain range is set as

$$global_{size} \leftarrow number_{ants} \tag{3}$$

The matrix of candidate solutions ($\text{solution}[\cdot][\cdot]$) stores the ants' paths, with each row representing a complete ant's solution. The set of visited nodes, $\text{visited}[\cdot]$, keeps track of the current visited nodes for each ant, preventing duplicate selection as forbidden by the TSP: the i-th element is set to *true* when the i-th node is chosen to be part of the ant's solution (initially all elements are set to *false*). At a current node c, $\text{selection}_{prob}[i]$ stores the probability of each node i being selected, which is based on the pheromone trails and heuristic information—such data is cached in $\text{choice}_{info}[c][i]$.

Algorithm 5: OpenCL kernel for the ant-based solution construction

```
// Initialization
visited[·] ← false;
// Selection of the initial node
Initial_node ← Random(0, number_nodes − 1);
solution[global_id × number_nodes + 0] ← Initial_node;
visited[global_id × number_nodes + Initial_node] ← true;
for step ← 1 to number_nodes − 1 do
    sum_prob ← 0.0;
    current_node ← solution[global_id × number_nodes + (step − 1)];
    // Calculation of the nodes' probabilities
    for i ← 0 to number_nodes − 1 do
        if visited[global_id × number_nodes + i] then
            selection_prob[global_id × number_nodes + i] ← 0.0;
        else
            selection_prob[global_id × number_nodes + i] ← choice_info[current_node × number_nodes + i];
            sum_prob ← sum_prob + selection_prob[global_id × number_nodes + i];
    // Node selection via roulette wheel
    r ← Random(0, sum_prob);
    i ← 0;
    p ← selection_prob[global_id × number_nodes + 0];
    while p < r do
        i ← i + 1;
        p ← p + selection_prob[global_id × number_nodes + i];
    solution[global_id × number_nodes + step] ← i;
    visited[global_id × number_nodes + i] ← true;
```

The function `Random(a,b)` returns a uniform real-valued pseudo-number between a and b. The random number generator could be easily implemented on the GPU through the simple linear congruential method [29]; the only requirement would be to keep in the device's global memory a *state* information (an integral number) for each work-item that must persist across kernel executions.

There exist data-based parallel strategies for the construction of the solutions, where usually a work-group takes care of an ant and its work-items compute in parallel some portion of the construction procedure. For instance, the ANT_{block} strategy in [27], which in parallel evaluates and chooses the next node (city) from all the possible candidates. However, those strategies are considerably more complex than the ant-based parallelism, and for large-scale problems in which the number of ants is reasonably high—i.e. the class of problems that one would make use of GPUs—the ant-based strategy is enough to saturate the GPU.

4.3. Solution evaluation

When all solutions are constructed, they must be evaluated. The direct approach is to parallelize this step by the number of ants, dedicating a work-item per solution. However, in many problems it is possible to decompose the evaluation of the solution itself, leading

http://dx.doi.org/10.5772/51679

to a second level of parallelism: each work-group takes care of an ant, with each work-item within this group in charge of a subset of the solution.

4.3.1. Ant-based Evaluation (AE)

The simplest strategy for evaluating the solutions is to parallelize by the number of ants, assigning each solution evaluation to a work-item. In this case, the kernel could be written as in Algorithm 6, with the 1-D domain range as

$$global_{size} \leftarrow number_{ants} \tag{4}$$

The cost resulting from the evaluation of the complete solution of ant k, which in the kernel

Algorithm 6: OpenCL kernel for the ant-based evaluation

$\text{solution}_{value}[global_{id}] \leftarrow 0.0$;

for $i \leftarrow 0$ **to** $number_{nodes} - 2$ **do**
  $j \leftarrow \text{solution}[global_{id} \times number_{nodes} + i]$;
  $h \leftarrow \text{solution}[global_{id} \times number_{nodes} + (i+1)]$;
  $\text{solution}_{value}[global_{id}] \leftarrow \text{solution}_{value}[global_{id}] + \eta[j \times number_{nodes} + h]$;

$j \leftarrow \text{solution}[global_{id} \times number_{nodes} + (number_{nodes} - 1)]$;
$h \leftarrow \text{solution}[global_{id} \times number_{nodes} + 0]$;
$\text{solution}_{value}[global_{id}] \leftarrow \text{solution}_{value}[global_{id}] + \eta[j \times number_{nodes} + h]$;

is denoted by $global_{id}$, is put into the array $\text{solution}_{value}[k]$ of dimension $number_{ants}$.

4.3.2. Data-based Evaluation (DE)

This second strategy adds one more level of parallelism than the one previously presented. In the case of TSP, the costs of traveling from node i to j, j to k and so on can be summed up in parallel. To this end, the parallel primitive known as *prefix sum* is employed [30]. Its idea is illustrated in Figure 3, where $w_0 \ldots w_{N-1}$ correspond to the work-items within a work-group. The computational step complexity of the parallel prefix sum is $O(log_2 N)$, meaning that, for instance, the sum of an array of 8 nodes is computed in just 3 iterations.

In order to apply this primitive to a TSP's solution, a preparatory step is required: the cost for each adjacent node must be obtained from the distance matrix and put into an array, let us call it δ.[11] This preprocessing is done in parallel, as shown in Algorithm 7, which also describes the subsequent prefix sum procedure. In the kernel, the helper function `Distance(`k, i`)` returns the distance between the node i and $i+1$ for ant k; when i is the last node, the function returns the distance from this one to the first node. One can notice the use of the function `Barrier()`. In OpenCL, a barrier is a synchronization point that ensures that a memory region written by other work-items is consistent at that point. The first barrier is necessary because $\delta[local_{id} - s]$ references a memory region that was written by the s-th previous work-item. As for the second barrier, it is needed to prevent $\delta[local_{id}]$ from being updated before the s-th next work-item reads it. Finally, the final sum, which ends up at the last element of δ, is stored in the solution_{value} vector for the ant indexed by $group_{id}$.

[11] To improve efficiency, the array δ could—and frequently is—be allocated directly in the *local* memory (cf. 2.1).

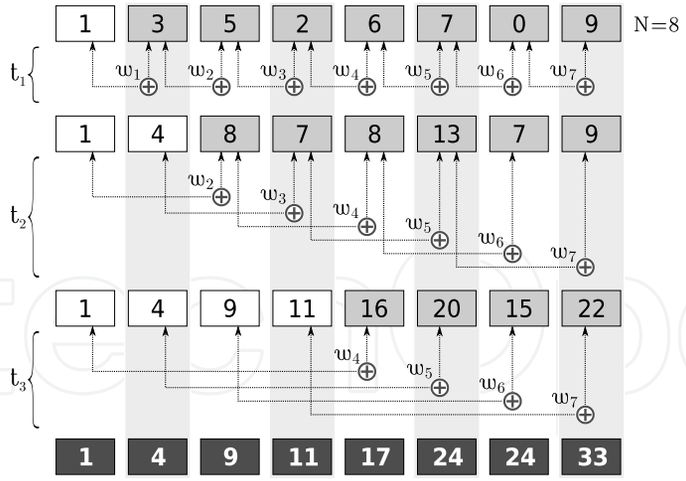

Figure 3. Parallel *prefix sum*: each element of the final array is the sum of all the previous elements, i.e. the partial cost; the last element is the total cost.

Algorithm 7: OpenCL kernel for the data-based evaluation

```
// Preparatory step
δ[local_id] ← Distance(group_id, local_id);
// Prefix sum
tmp ← δ[local_id];
s ← 1;
while s < local_size do
    Barrier();
    if local_id ≥ s then
        tmp ← δ[local_id] + δ[local_id − s];
    Barrier();
    δ[local_id] ← tmp;
    s ← s × 2;
if local_id = group_size − 1 then
    solution_value[group_id] ← δ[group_size − 1];
```

Regarding the N-D domain definition, since there are $number_{ants}$ ants and for each ant (solution) there are $number_{nodes}$ distances, the global size is given by

$$global_{size} \leftarrow number_{ants} \times number_{nodes} \tag{5}$$

and the local size, i.e. the amount of work-items devoted to compute the total cost per solution, simply by

$$local_{size} \leftarrow number_{nodes}, \tag{6}$$

resulting in $number_{ants}$ work-groups (one per ant).[12]

[12] For the sake of simplicity, it is assumed that the number of nodes (cities) is such that the resulting local size is less than the device's maximum supported local size, a hardware limit. If this is not the case, then Algorithm 7 should be modified in such a way that each work-item would compute more than just one partial sum.

http://dx.doi.org/10.5772/51679

4.3.3. Finding the Best Solution

It is important at each iteration to keep track of the best-so-far solution. This could be achieved naively by iterating over all the evaluated solutions sequentially. There is though a parallel alternative to that which utilizes a primitive, analogous to the previous one, called *reduction* [30]. The idea of the parallel reduction is visualized in Figure 4. It

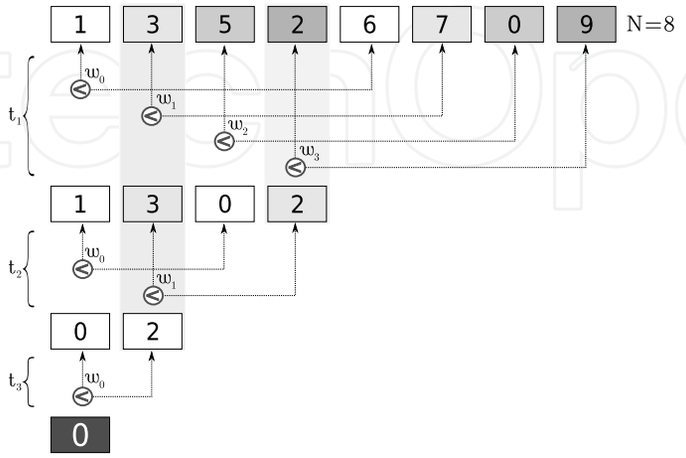

Figure 4. $O(log_2 N)$ parallel reduction: the remaining element is the smallest of the array.

starts by comparing the elements of an array—that is, solution$_{value}$—by pairs to find the smallest element between each pair. The next iteration finds the smallest values of the previously reduced ones, then the process continues until a single value remains; this is the smallest element—or cost—of the entire array. The implementation is somewhat similar to the prefix sum, and will not be detailed here. The global and local sizes should both be set to $number_{ants}$, meaning that the reduction will occur within one work-group since synchronization is required. The actual implementation will also need a mapping between the cost values (the solution$_{value}$ array) and the corresponding solutions in order to link the smallest cost found with the respective solution.

4.4. Pheromone Trails Updating

After all ants have constructed their tours (solutions), the pheromone trails are updated. In AS, the pheromone update step starts evaporating *all* arcs by a constant factor, followed by a reinforcement on the arcs visited by the ants in their tours.

4.4.1. Pheromone Evaporation

In the pheromone evaporation, each element of the pheromone matrix has its value decreased by a constant factor $\rho \in (0, 1]$. Hence, the parallel implementation can explore parallelism in the order of $number_{nodes} \times number_{nodes}$. For this step, the kernel can be described as in Algorithm 8, with the 2-D domain range given by

$$\begin{aligned} global^0_{size} &\leftarrow number_{nodes} \\ global^1_{size} &\leftarrow number_{nodes} \end{aligned} \tag{7}$$

Algorithm 8: OpenCL kernel for computing the pheromone evaporation

$\tau[global^0_{id} \times global^1_{size} + global^1_{id}] \leftarrow (1 - \rho) \times \tau[global^0_{id} \times global^1_{size} + global^1_{id}];$

4.4.2. Pheromone Updating

After evaporation, ants deposit different quantities of pheromone on the arcs that they crossed. Therefore, in an ant-based parallel implementation each element of the pheromone matrix may potentially be updated by many ants at the same time, leading of course to memory inconsistency. An alternative is to parallelize on the ant's solution, taking advantage of the fact that in the TSP there is no duplicate node in a given solution. This strategy works on one ant k at a time, but all edges (i, j) are processed in parallel. Hence, the 1-D domain range is given by

$$global_{size} \leftarrow number_{nodes} - 1, \tag{8}$$

with the corresponding kernel described in Algorithm 9. The kernel should be launched $number_{ants}$ times from the host code, each time passing a different $k \in [0, number_{ants})$ as a kernel's argument—the only way of guaranteeing global memory consistency (synchronism) in OpenCL, which is necessary to prevent two or more ants from being processed simultaneously, is when a kernel finishes its execution.

Algorithm 9: OpenCL kernel for updating the pheromone for ant k

$i \leftarrow \text{solution}[k \times number_{nodes} + global_{id}];$
$j \leftarrow \text{solution}[k \times number_{nodes} + global_{id} + 1];$

$\tau[i \times number_{nodes} + j] \leftarrow \tau[i \times number_{nodes} + j] + 1.0/\text{solution}_{value}[k];$
$\tau[j \times number_{nodes} + i] \leftarrow \tau[i \times number_{nodes} + j];$

5. Conclusions

This chapter has presented and discussed different parallelization strategies for implementing an Ant Colony Optimization algorithm on Graphics Processing Unit, presenting also a list of references on previous works on this area.

The chapter also provided straightforward explanation of the GPU architecture and gave special attention to the Open Computing Language (OpenCL), explaining in details the concepts behind these two topics, which are often just mentioned in references in the literature.

It was shown that each step of an ACO algorithm, from the initialization phase through the return of the final solution, can be parallelized to some degree, at least at the granularity of the number of ants. For complex or large-scale problems—in which numerous ants would be desired—the ant-based parallel strategies should suffice to fully explore the computational power of the GPUs.

Although the chapter has focused on a particular computing architecture, the GPU, all the described kernels can be promptly executed on any other OpenCL parallel device, such as the multi-core CPUs.

http://dx.doi.org/10.5772/51679

Finally, it is expected that this chapter will provide the readers with an extensive view of the existing ACO parallel strategies on the GPU and will assist them in developing new or derived parallel strategies to suit their particular needs.

Acknowledgments

The authors thank the support from the Brazilian agencies CNPq (grants 141519/2010-0 and 308317/2009-2) and FAPERJ (grant E-26/102.025/2009).

Author details

Jaqueline S. Angelo[1,*],
Douglas A. Augusto[1] and Helio J. C. Barbosa[1,2]

* Address all correspondence to: jsangelo@lncc.br; douglas@lncc.br; hcbm@lncc.br

1 Laboratório Nacional de Computação Científica (LNCC/MCTI), Petropólis, RJ, Brazil
2 Universidade Federal de Juiz de Fora (UFJF), MG, Brazil

References

[1] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, 1992.

[2] Marco Dorigo and Thomas Stutzle. *Ant Colony Optimization*. The MIT Press, 2004.

[3] Thomas Stutzle. Parallelization strategies for ant colony optimization. In *Proc. of PPSN-V, Fifth International Conference on Parallel Problem Solving from Nature*, pages 722–731. Springer-Verlag, 1998.

[4] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. A survey on parallel ant colony optimization. *Appl. Soft Comput.*, 11(8):5181–5197, December 2011.

[5] Stefan Janson, Daniel Merkle, and Martin Middendorf. *Parallel Ant Colony Algorithms*, pages 171–201. John Wiley and Sons, Inc., 2005.

[6] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. *Swarm Intelligence*. Oxford University Press, Oxford, New York, 1999.

[7] Marco Dorigo, Eric Bonabeau, and Guy Theraulaz. Ant algorithms and stigmergy. *Future Gener. Comput. Syst.*, 16(9):851–871, 2000.

[8] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5:137–172, 1999.

[9] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: Optimization by a colony of cooperating agents. *IEEE Trans. on Systems, Man, and Cybernetics–Part B*, 26(1):29–41, 1996.

[10] R.J. Mullen, D. Monekosso, S. Barman, and P. Remagnino. A review of ant algorithms. *Expert Systems with Applications*, 36(6):9608 – 9617, 2009.

[11] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2011.

[12] Ethan Mollick. Establishing Moore's law. *IEEE Ann. Hist. Comput.*, 28:62–75, July 2006.

[13] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[14] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems.

[15] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, November 2011.

[16] Douglas A. Augusto and Helio J.C. Barbosa. Accelerated parallel genetic programming tree evaluation with opencl. *Journal of Parallel and Distributed Computing*, (0):–, 2012.

[17] Advanced Micro Devices. *AMD Accelerated Parallel Processing Programming Guide - OpenCL*, 12 2010.

[18] NVIDIA Corporation. *OpenCL Best Practices Guide*, 2010.

[19] A. Catala, J. Jaen, and J.A. Modioli. Strategies for accelerating ant colony optimization algorithms on graphical processing units. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 492 –500, 2007.

[20] Hongtao Bai, Dantong OuYanga, Ximing Li, Lili He, and Haihong Yu. MAX-MIN ant system on GPU with CUDA. In *Fourth International Conference on Innovative Computing, Information and Control*, pages 801–804, 2009.

[21] Weihang Zhu and James 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*, SMC'09, pages 1803–1808. IEEE Press, 2009.

[22] Jie Fu, Lin Lei, and Guohua Zhou. A parallel ant colony optimization algorithm with gpu-acceleration based on all-in-roulette selection. In *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, pages 260–264, 2010.

[23] Jose A. Mocholi, Javier Jaen, Alejandro Catala, and Elena Navarro. An emotionally biased ant colony algorithm for pathfinding in games. *Expert Systems with Applications*, 37:4921–4927, 2010.

[24] Nicholas A. Sinnott-Armstrong, Casey S. Greene, and Jason H. Moore. Fast genome-wide epistasis analysis using ant colony optimization for multifactor dimensionality reduction analysis on graphics processing units. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO 2010, pages 215–216, New York, NY, USA, 2010. ACM.

http://dx.doi.org/10.5772/51679

[25] S. Tsutsui and N. Fujimoto. Fast qap solving by aco with 2-opt local search on a gpu. pages 812 –819, june 2011.

[26] Min Li, Kelson Gent, and Michael S. Hsiao. Utilizing gpgpus for design validation with a modified ant colony optimization. *High-Level Design, Validation, and Test Workshop, IEEE International*, 0:128–135, 2011.

[27] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki. Parallel ant colony optimization on graphics processing units. *J. Parallel Distrib. Comput.*, 2012.

[28] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on gpus. *Journal of Parallel and Distributed Computing*, 2012.

[29] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, 3 edition, November 1997.

[30] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

IntechOpen