Enhancing Data Parallelism for Ant Colony Optimisation on GPUs

José M. Cecilia*, José M. García
Computer Architecture Department. University of Murcia. 30100 Murcia, Spain

Andy Nisbet, Martyn Amos
Novel Computation Group. Division of Computing and IS. Manchester Metropolitan University. Manchester M1 5GD, UK

Manuel Ujaldón
Computer Architecture Department. University of Malaga. 29071 Málaga, Spain

Abstract

Graphics Processing Units (GPUs) have evolved into highly parallel and fully programmable architectures over the past five years, and the advent of CUDA has facilitated their application to many real-world applications. In this paper, we deal with a GPU implementation of Ant Colony Optimisation (ACO), a population-based optimisation method which comprises two major stages: *Tour construction* and *Pheromone update*. Because of its inherently parallel nature, ACO is well-suited to GPU implementation, but it also poses significant challenges due to irregular memory access patterns. Our contribution within this context is threefold: (1) a data parallelism scheme for *Tour construction* tailored to GPUs, (2) novel GPU programming strategies for the *Pheromone update* stage, and (3) a new mechanism called I-Roulette to replicate the classic Roulette Wheel while improving GPU parallelism. Our implementation leads to factor gains exceeding 20x for any of the two stages of the ACO algorithm as applied to the TSP when compared to its sequential counterpart version running on a similar single-threaded high-end CPU. Moreover, an extensive discussion focused on different implementation paths on GPUs shows the way to deal with parallel graph connected components. This, in turn, suggests a broader area of enquiry, where algorithm designers may learn to adapt similar optimisation methods to GPU architecture.

*Corresponding author

Email addresses: chema@ditec.um.es (José M. Cecilia), jmgarcia@ditec.um.es (José M. García), a.nisbet@mmu.ac.uk (Andy Nisbet), m.amos@mmu.ac.uk (Martyn Amos), ujaldon@uma.es (Manuel Ujaldón)

Keywords: Metaheuristics, GPU programming, Ant Colony Optimization, TSP, Performance Analysis

1. Introduction

Ant Colony Optimisation (ACO) [1] is a population-based search method inspired by the behaviour of real ants. It may be applied to a wide range of problems [2, 3], many of which are graph-theoretic in nature. It was first applied to the Traveling Salesman Problem (TSP) [4] by Dorigo *et al.* [5, 6].

In essence, simulated ants construct solutions to the TSP in the form of *tours*. The artificial ants are simple agents which construct tours in a parallel, probabilistic fashion. They are guided in this task by simulated *pheromone trails* and *heuristic information*. Pheromone trails are a fundamental component of the algorithm, since they facilitate indirect communication between agents via their *environment*, a process known as *stigmergy* [7]. For additional details about these processes, we refer the reader to [1].

ACO algorithms are population-based, that is, a collection of agents "collaborates" to find an optimal (or at least satisfactory) solution. Such approaches are suited to parallel processing, but their success strongly depend on the nature of the particular problem and the underlying hardware available. Several parallelisation strategies have been proposed for the ACO algorithm on shared and distributed memory architectures [8, 9, 10].

The *Graphics Processing Unit* (GPU) is a topic of significant interest in high performance computing. For applications with abundant parallelism, GPUs deliver higher peak computational throughput than latency-oriented CPUs, thus offering a tremendous potential performance uplift on massively parallel problems [11]. Of particular relevance to us are attempts to parallelise the ACO algorithm on GPUs. Until now, these approaches have focussed on accelerating the *tour construction* step performed by each ant by taking a task-based parallelism approach, with pheromone deposition on the CPU [12, 13, 14].

In this paper, we present the first fully developed ACO algorithm for the Traveling Salesman Problem (TSP) on GPUs where *both* stages are parallelised: *Tour construction* and *Pheromone update*. A *data parallelism* approach, which is better suited to GPU parallelism model, is used to enhance performance on the first stage, and several GPU design patterns are evaluated for the parallelisation of the second stage. Our major contributions include the following:

1. To the best of our knowledge, this is the first data-parallelism scheme on GPUs for the ACO tour construction stage. Our design proposes two different types of virtual ants: *Queen* ants (associated with CUDA thread-blocks), and *worker* ants (associated with CUDA threads).
2. We introduce an *I-Roulette* method (Independent Roulette) to replicate the classic Roulette Wheel selection while improving GPU parallelism.
3. We discuss the implementation of the pheromone update stage on GPUs, using either atomic operations or other GPU alternatives.
4. We offer an in-depth analysis of both stages of the ACO algorithm for different instances of the TSP problem. Several GPU parameters are tuned to reach a speed-up factor of up to 21x for the tour construction stage, and a 20x speed-up factor for the pheromone update stage.
5. The solution accuracy obtained by our GPU algorithms is compared to that of the sequential code given in [1] and extended using TSPLIB.

The rest of the paper is organised as follows. We briefly introduce Ant Colony Optimisation for the TSP and Compute Unified Device Architecture (CUDA) from NVIDIA in Section 2. In Section 3 we present GPU designs for the main stages of the ACO algorithm. Our experimental methodology is outlined in Section 4 before we describe the performance evaluation of our algorithm in Section 5. Other parallelization strategies for the ACO algorithm are described in Section 6, before we summarise our findings and conclude with suggestions for future work.

2. Background

2.1. Ant Colony Optimisation for the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) [4] involves finding the shortest (or "cheapest") round-trip route that visits each of a number of "cities" exactly once. The symmetric TSP on n cities may be represented as a complete weighted graph, G, of n nodes, with each weighted edge, $e_{i,j}$, representing the inter-city distance $d_{i,j} = d_{j,i}$ between cities i and j. The TSP is a well-known NP-hard optimisation problem, and is used as a standard benchmark for many heuristic algorithms [15].

The TSP was the first problem solved by Ant Colony Optimisation (ACO) [6, 16]. This method uses a number of simulated "ants" (or *agents*), which perform distributed search on a graph. Each ant moves on the graph until it completes a tour, and then offers this tour as its suggested solution. In

order to achieve this latter step, each ant drops "pheromone" on the edges that it visits during its tour. The quantity of pheromone dropped, if any, is determined by the *quality* of the ant's solution relative to those obtained by the other ants. The ants probabilistically choose the next city to visit, based on *heuristic information* obtained from inter-city distances and the net pheromone trail. Although such heuristic information drives the ants towards an optimal solution, a process of pheromone "evaporation" is also applied in order to prevent the process stalling in a local minimum.

The Ant System (AS) is an early variant of ACO, first proposed by Dorigo [16]. The AS algorithm is divided into two main stages: *Tour construction* and *Pheromone update*. Tour construction is based on m ants building tours in parallel. Initially, ants are randomly placed. At each construction step, each ant applies a probabilistic action choice rule, called the *random proportional rule*, which decides the city to visit next. The probability for ant k, placed at city i, of visiting city j is given by the equation 1

$$p_{i,j}^k = \frac{[\tau_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{l \in N_i^k} [\tau_{i,l}]^\alpha [\eta_{i,l}]^\beta}, \qquad if\ j \in N_i^k, \tag{1}$$

where $\eta_{i,j} = 1/d_{i,j}$ is a heuristic value determined *a priori*, α and β are two parameters determining the relative *influences* of the pheromone trail and the heuristic information respectively, and N_i^k is the feasible neighborhood of ant k when at city i. This latter set represents the set of cities that ant k has not yet visited; the probability of choosing a city outside N_i^k is zero (this prevents an ant returning to a city, which is not allowed in the TSP). By this probabilistic rule, the probability of choosing a particular edge (i, j) increases with the value of the associated pheromone trail $\tau_{i,j}$ and of the heuristic information value $\eta_{i,j}$. The numerator of the equation 1 is the same for every ant in a single run, which encourages efficiency by storing this information in an additional matrix, called *choice_info* (see [1]). The random proportional rule ends with a selection procedure, which is done analogously to the *roulette wheel* selection procedure of evolutionary computation (see [1], [17]). Each value $choice\_info[current\_city][j]$ of a city j that ant k has not yet visited is associated with a slice on a circular roulette wheel, with the size of the slice being proportional to the weight of the associated choice. The wheel is then "spun", and the city to which a fixed marker points is chosen as the next city for ant k. Additionally, each ant k maintains a memory, M^k, called the *tabu list*, which contains a chronological ordering of the cities

already visited. This memory is used to determine the feasible neighborhood, and also allows an ant to (1) compute the length of the tour T^k it generated, and (2) retrace the path to deposit pheromone.

After all ants have constructed their tours, the pheromone trails are updated. This is achieved by first lowering the pheromone value on all edges by a constant factor (analogous to evaporation), and then adding pheromone to edges that ants have crossed in their tours. Pheromone evaporation is implemented by

$$\tau_{i,j} \leftarrow (1-\rho)\tau_{i,j}, \qquad \forall (i,j) \in L, \tag{2}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. After evaporation, all ants deposit pheromone on their visited edges:

$$\tau_{i,j} \leftarrow \tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}, \qquad \forall (i,j) \in L, \tag{3}$$

where $\Delta\tau_{ij}$ is the amount of pheromone ant k deposits. This is defined as follows:

$$\Delta\tau_{i,j}^{k} = \begin{cases} 1/C^k \text{ if } e(i,j)^k & \text{belongs to } T^k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

where C^k, the length of the tour T^k built by the k-th ant, is computed as the sum of the lengths of the edges belonging to T^k. According to equation 4, the better an ant's tour, the more pheromone the edges belonging to this tour receive. In general, edges that are used by many ants (and which are part of short tours), receive more pheromone, and are therefore more likely to be chosen by ants in future iterations of the algorithm.

2.2. CUDA Programming model

All Nvidia GPU platforms from the G80 architecture may be programmed using the Compute Unified Device Architecture (CUDA) programming model, which makes GPUs operate as a highly parallel computing device. Each GPU device is a scalable processor array consisting of a set of SIMT (Single Instruction Multiple Threads) Streaming Multiprocessors (SM), each containing several stream processors (SPs). Different memory spaces are available in each GPU on the system. The global memory (also called *device* or video memory) is the only space accessible to all multiprocessors. It is the largest

(and slowest) memory space, and is private to each GPU on the system. Additionally, each *multiprocessor* has its own private memory space, called *shared memory*. The shared memory is smaller than global memory, with lower access latency. Finally, there exist other addressing spaces that are dedicated to specific purposes, such as texture and constant memory [18].

The CUDA programming model is based on a hierarchy of abstraction layers The **thread** is the basic execution unit that is mapped to a single SP. A **thread-block** is a batch of threads which can cooperate, (as they are assigned to the same multiprocessor) and therefore share all the resources included in that multiprocessor, such as the register file and shared memory. A **grid** is composed of several thread-blocks which are equally distributed and scheduled across all multiprocessors in a non-deterministic manner. Finally, threads included within a thread-block are divided into batches of 32 threads called **warps**. The warp is the scheduled unit, so the threads of the same thread-block are executed in a given multiprocessor warp-by-warp in a SIMD fashion (same instruction over multiple data). The programmer arranges parallelism by declaring the number of thread-blocks, the number of threads per thread-block and their distribution, subject to the program constraints (i.e., data and control dependencies).

3. Code Design and Tuning Techniques

Algorithm 1 The sequential AS version for the TSP:

```
1: InitializeData()
2: while ¬Convergence() do
3:    TourConstruction()
4:    PheromoneUpdate()
5: end while
```

In this Section, we present several different GPU designs for the Ant System (AS), as applied to the TSP. Algorithm 1 shows Single Program Multiple Data (SPMD) pseudocode for the AS. Firstly, all AS structures for the TSP problem (distance matrix, number of cities, ...) are initialized. Next, the *Tour construction* and *Pheromone update* stages are performed until the convergence criterion is reached. For tour construction, we begin by analysing CPU alternatives and traditional implementations based on *task* parallelism on GPUs, which motivates our approach of increasing *data* parallelism in-

stead. For pheromone update, we describe several GPU techniques that are useful for increasing the data bandwidth in this application.

3.1. Previous tour construction proposals

3.1.1. CPU baseline

The tour construction stage is divided into two stages: *Initialization* and *ASDecisionRule*. In the former, all data structures (tabu list, initial random city, ...) are initialized by each ant. Algorithm 2 shows the latter stage, which is further divided into two sub-stages. First, each ant calculates the heuristic information to visit city j from city i according to equation 1 (lines 1-11). As previously explained, it is computationally expensive to repeatedly calculate these values for each computational step of each ant, k, and this can be avoided by using an additional data structure, *choice_info*, in which those heuristic values are stored using an adjacency matrix [1]. We note that each entry in this structure may be calculated *independently* (see Eq. 1).

After the heuristic information has been calculated, the probabilistic choice of next city for each ant is calculated using roulette wheel selection [1, 17] (see Algorithm 2, lines 12-18).

3.1.2. Task parallelism approach on GPUs

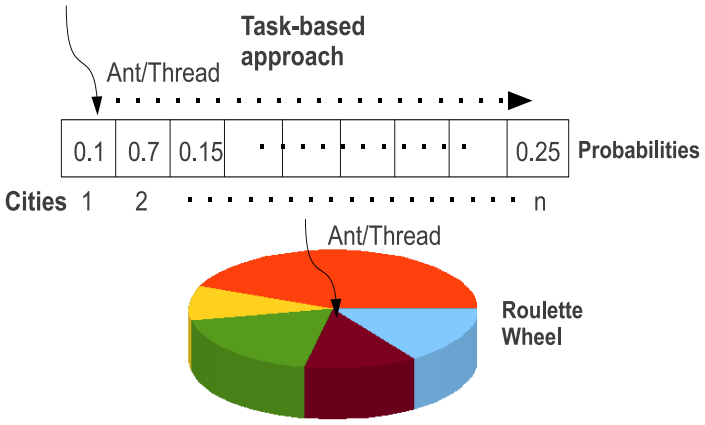

Figure 1: Task parallelism on the tour construction kernel.

The "traditional" task parallelism approach to tour construction is based on the observation that ants run in parallel looking for the best tour they can find. Therefore, any inherent parallelism exists at the level of individual ants. To implement this idea of parallelism using CUDA, each ant is identified as

Algorithm 2 *ASDecisionRule* for the Tour construction stage. m is the number of ants, and n is the number of cities in the TSP instance

```
1: sum_prob ← 0.0;
2: current_city ← ant[k].tour[step − 1];
3: for j = 1 to n do
4:    if ant[k].visited[j] then
5:       selection_prob[j] ← 0.0;
6:    else
7:       current_probability ← choice_info[current_city][j];
8:       selection_prob[j] ← current_probability;
9:       sum_probs ← sum_probs + current_probability;
10:   end if
11: end for
    {Roulette Wheel Selection Process}
12: r ← random(1..sum_probs);
13: j ← 1;
14: p ← selection_prob[j];
15: while p < r do
16:   j ← j + 1;
17:   p ← p + selection_prob[j];
18: end while
19: ant[k].tour[step] ← j;
20: ant[k].visited[j] ← true;
```

a CUDA thread, and threads are equally distributed among CUDA thread blocks. Each thread deals with the task assigned to each ant; i.e, maintenance of an ant's memory (list of all visited cities, and so on) and movement (see the core of this computation in Algorithm 2). Figure 1 briefly summarizes the process sequentially developed by each ant.

To improve the application bandwidth, some data structures may be placed in on-chip shared memory. Of these, *visited* and *selection_prob* list are good candidates to be placed on shared memory as they are accessed many times during the computation, in an irregular access pattern. However, shared memory is a scarce resource in CUDA capable GPUs [18], and thus the size of these structures is naturally limited. Moreover, in the CUDA programming model, shared memory is allocated at CUDA thread block level.

3.2. Our tour construction approach based on data parallelism

The task parallelism approach is challenging for GPUs for several reasons. Firstly, it requires a relatively low number of threads on the GPU (the suggested number of ants for solving the TSP problem matches the number of cities [1]). Secondly, this version presents an unpredictable memory access pattern, due to its execution being guided by a stochastic process. Finally, the checking of the list of cities visited contains many warp divergences (threads within a warp taking different paths), leading to serialisation [18].

3.2.1. The choice_info matrix calculation on GPU

In order to increase parallelism in CUDA, the *choice_info* computation is performed apart from the tour construction kernel, being included in a different CUDA kernel which is executed right before the tour construction. We set a CUDA thread for each entry of the *choice_info* structure, and these are equally grouped into CUDA thread blocks. However, the performance for this kernel may be drastically affected by the use of a costly math function like *powf()* (see Equation 1). Fortunately, there are analogous CUDA functions which map directly to the hardware level (like *__powf()*), although this comes at the expense of some loss of accuracy [19].

3.2.2. Data parallelism approach

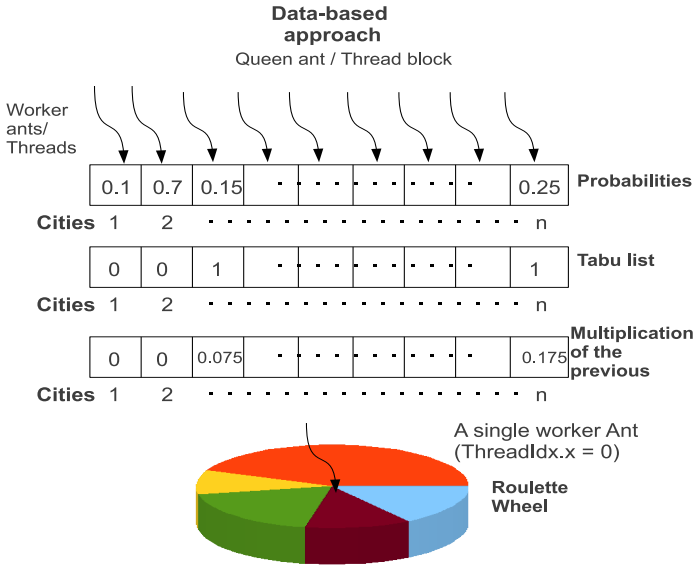

Figure 2: Data parallelism approach on the tour construction kernel.

Figure 2 shows an alternative design, which increases *data-parallelism* in the tour construction kernel, and also avoids warp divergences. In this design, we propose the use of two different types of virtual ants to use an unmistakable metaphor. *Queen ants* represent the simulated ants, and *Worker ants* collaborate with each queen to accelerate the decision about the next city to visit. Thus, each queen has her own group of workers to explore paths more quickly.

A *thread-block* is associated with each queen ant, and each thread within a block represents a city (or cities) a worker ant may visit. All w worker ants cooperate to obtain a solution, increasing data-parallelism a factor of w.

A thread loads the heuristic value associated with each associated city, and checks whether or not the city has been visited. To avoid conditional statements (and, thus, warp divergences), the tabu list (i.e., the list of cities that *may not* be visited) is represented in shared memory as one integer value per city. A city's value in this list is 0 if it has been visited, and 1 otherwise. Finally, these values are multiplied and stored in a shared memory array, which is then prepared for the selection process via the simulated roulette wheel. We note that the shared memory requirements for this method are drastically reduced compared to those of the previous version. Now, the tabu list and the probabilistic list are only stored once per thread-block (i.e., Queen ant) instead of once per thread (i.e., Worker ant).

The number of threads per thread-block is an internal CUDA constraint (which evolves with the Compute Capabilities version installed on a given software version and hardware generation). Therefore, cities should ideally be distributed among threads in order to allow for a flexible implementation. A *tiling* technique is proposed to deal with this issue. Cities are divided into blocks (i.e., tiles). For each tile, a city is selected stochastically, from the set of unvisited cities on that tile. When this process is over, we have a set of "partial best" cities. Finally, the city with the best *absolute* heuristic value is selected from this partial best set.

The tabu list may be placed in the register file (since it represents information private to each thread). However, the tabu list cannot be represented by a single integer register per thread in the tiling version, because, in that case, a thread represents more than one city. The 32-bit registers may be used on a bitwise basis for managing the list. The first city represented by each thread, i.e., on the first tile, is managed by bit 0 on the register that represents the tabu list, the second city is managed by bit 1, and so on.

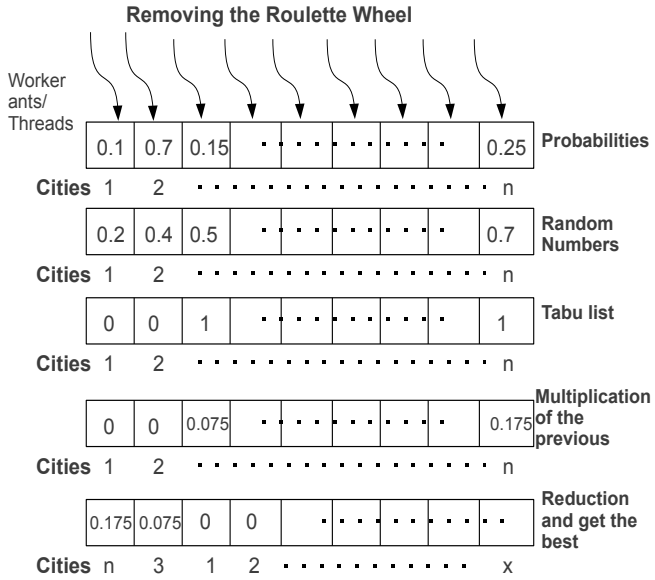

Figure 3: An alternative method for increasing the parallelism on the selection process.

3.2.3. I-Roulette: An alternative selection method

The roulette wheel is a fully sequential stochastic selection process, and, as such, is hard to parallelize. To implement this in CUDA, a particular thread is identified to proceed sequentially with the selection, doing this exactly $n-1$ times, with n being the number of cities. Moreover, the kernel has to generate costly pseudorandom numbers on the GPU, which we implement using Nvidia's CURAND library [20].

Figure 3 shows an alternative method for removing the sequential parts of the previous kernel design. We call this method *I-Roulette* (Independent Roulette). I-Roulette proceeds by generating a random number per city in the interval $[0, 1]$, which feeds into the stochastic simulation. Thus, three values are multiplied and stored in the shared memory array per city; i.e., the heuristic value associated with a city, a value showing whether the city has been visited or not, and the random number associated with a city. Finally, a reduction is performed to stochastically select the city to go (see Algorithm 3).

3.3. Pheromone update stage

The final stage in the ACO algorithm is pheromone update, which comprises two main tasks: pheromone evaporation and pheromone deposit. The first step is quite straightforward to implement in CUDA, as a single thread

Algorithm 3 *I-Roulette* method. It assumes this code is executed by as many threads as cities. We launch a random number for each thread.

1: $r \leftarrow random(seed)$;
2: $p \leftarrow selection\_prob[j]$;
3: $v = ant[k].visited[j]$; {Tabu list is (0 = visited); (1 = non-visited)}
4: $array[threadId] = r * p * v$;
5: $threadId\_best = Reduction(array)$;

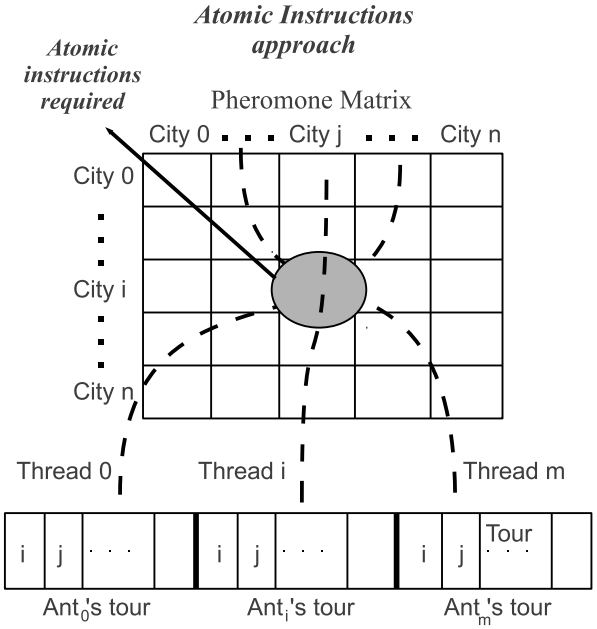

Figure 4: Pheromone deposit with atomic instructions.

can independently calculate the Equation 2 for each entry of the pheromone matrix, thus lowering the pheromone value on all edges by a constant factor.

Ants then deposit different quantities of pheromone on the edges that they have crossed in their tours. As stated previously, the quantity of pheromone deposited by each ant depends on the quality of the tour found by that ant (see Equations 3 and 4). Figure 4 shows the design of the pheromone kernel; this allocates a thread per city in an ant's tour. Each ant generates its own private tour in parallel, and they may visit the same edge as another ant. This fact forces us to use *atomic* instructions for accessing the pheromone matrix, leading to performance degradation. An alternative approach is shown in Figure 5, where we use a *scatter* to gather transformations [21].

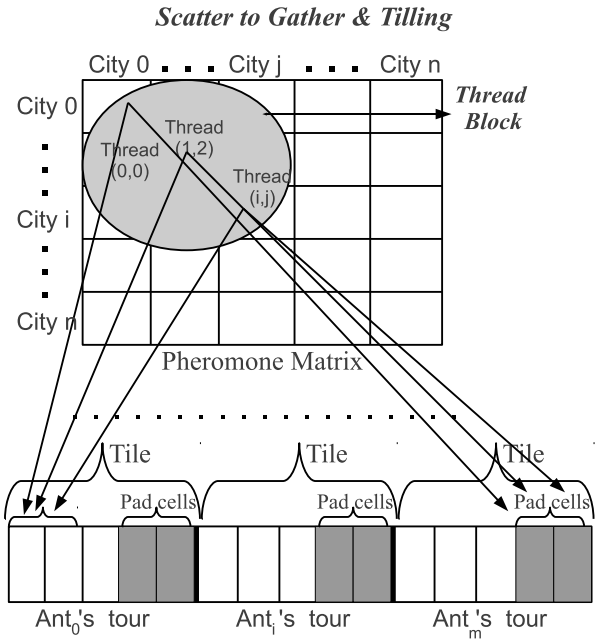

Figure 5: Scatter to Gather transformation for the pheromone deposit.

The configuration launch routine for the pheromone update kernel now creates as many threads as there are cells in the pheromone matrix ($c = n^2$), and equally distributes these threads among thread blocks. Thus, each thread represents a single entry in the pheromone matrix, and it is responsible for checking whether the cell that it represents has been visited by any ant. Each thread accesses device memory to check that information, which results in $2 * n^2$ memory loads per thread, for a total of $l = 2 * n^4$ (n^2 threads) accesses to device memory.

At this point, we have a tradeoff between the pressure on device memory to avoid a design based on atomic operations, and the number of atomic operations involved (relation $loads : atomic$ from now on). For the Scatter to Gather based design, the relation $loads : atomic$ is $l : c$. Therefore, this approach allows us to perform the computation whilst *removing* atomic operations, though this comes at the expense of drastically increasing the pressure on device memory. A tiling technique is thus proposed for increasing the application bandwidth. Now, all threads cooperate to load data from global memory to shared memory, but they still access edges in the ant's tour. Each thread accesses global memory $2n^2/\theta$, θ being the tile size. Any

remaining accesses are performed on shared memory, and the total number of global memory accesses is $\gamma = 2n^4/\theta$. The relation $loads : atomics$ is lower, $\gamma : c$, but it keeps on a similar order of magnitude.

An ant's tour length (i.e., $n+1$) may be larger than the maximum number of threads that each block can support [18]. Our algorithm prevents this situation by setting our empirically demonstrated optimum thread block layout, and then dividing the tour into tiles of this length. This raises another issue when $n+1$ is not divisible by θ. We solve this by applying padding to the ant tour array in order to avoid warp divergence (see Figure 5).

Unnecessary loads to device memory can be avoided by taking advantage of the symmetric version of the TSP, so the number of threads can be divided by two, thus halving the number of device memory accesses. This so-called Reduction version reduces the overall number of accesses to either shared or device memory and also applies tiling. The number of accesses per thread remains the same, for a total number of device memory access of $\rho = n_4/\theta$.

4. Experimental methodology

Table 1: Hardware features for the Intel Xeon CPU and the Nvidia Tesla C2050 GPU we have used for running our experiments.

	CPU	GPU
Manufacturer	Intel	Nvidia
Model	Xeon E5620	Tesla C2050
Codename/architecture	Westmere	Fermi
Clock frequency	2.4 GHz	1.15 GHz
L1 cache size	32KB + 32KB	16KB (+ 48 KB SM)
L2 cache size	256 KB	768 KB.
L3 cache size	12 MB	Does not have
DRAM memory	16 GB. DDR3	3 GB. GDDR5

During our experimental study, we have used the following platforms:

- **On the CPU side:** An Intel Xeon E5620 Westmere processor running at 2.40 GHz and endowed with four cores and 16 Gbytes of DDR3 memory.
- **On the GPU side:** A Nvidia Tesla C2050 Fermi graphics card endowed with 448 streaming processors and 3 GB of GDDR5 video memory.

For further features about these two platforms, see Table 1. Moreover, we use gcc 4.3.4 with the -O3 flag to compile our CPU implementations, and CUDA compilation tools release 3.2 on the GPU side.

4.1. Benchmarking

We test our algorithms using a standard set of benchmark instances from the well-known TSPLIB library [22] [23]. All benchmark instances are defined on a complete graph, and all distances are defined as integers. Table 2 shows all benchmark instances used, with information on the number of cities and the length of their optimal tour. ACO parameters such as the number of ants m, α, β, ρ, and so on are set according to the values recommended in [1] in order to focus on the parallelization side of the algorithm. Key parameters for the purpose of this study are the number of ants, which is set $m = n$ (n being the number of cities), $\alpha = 1$, $\beta = 2$, and $\rho = 0.5$.

Table 2: Summary of major features in our benchmark instances taken from TSPLIB library. "Cities" is the number of cities in the graph, and "Length" is the best tour length, that is, the minimum solution found based on 2D euclidean distance.

Name	d198	a280	lin318	pcb442	rat783	pr1002	pcb1173	d1291	pr2392
Cities	198	280	318	442	783	1002	1173	1291	2392
Length	15780	2579	42029	50778	8806	259045	56892	50801	378032

5. Performance Evaluation

This section analyses the two major stages of the ACO algorithm: *Tour construction* and *Pheromone update*. We compare our implementations to sequential code, written in ANSI C, provided by Stüzle in [1]. Performance figures are given for single-precision numbers and a single iteration run averaged over 100 iterations. We focus on the computational features of the Ant System and how it can be efficiently implemented on GPUs, but to guarantee the correctness of our algorithms, a quality comparison between the results obtained by the sequential and GPU codes is also provided.

5.1. Evaluation of tour construction stage

First, we evaluate the tour construction stage on Tesla C2050 from different perspectives: The performance impact of using costly arithmetic instruction in the *choice_info* kernel, comparison versus a single-threaded CPU counterpart version, improvement through a data parallelism approach, speed-up factor obtained when using different on-chip GPU memories, and performance benefits of changing the selection process. We now address each of these issues separately.

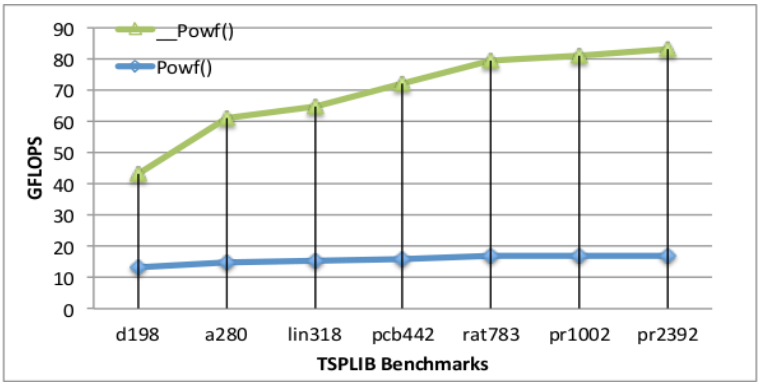

Figure 6: Giga FLoating Point Operations per Second (GFLOPS) on the Tesla C2050 GPU system for the *choice_info* kernel when using CUDA instructions *powf* and *__powf*.

5.1.1. Choice_info kernel evaluation

We first evaluate the *choice_info* kernel, before assessing the impact of several modifications to tour construction. Figure 9 shows performance figures, which are affected by using costly math functions like *powf()*. To reduce this overhead we chose an analogous CUDA function, *__powf()*, which is mapped directly to the hardware level [19]. This is faster, because it does not check for special cases, which are irrelevant to our computational case. This way, loss of accuracy is negligible, but the observed performance gain is remarkable.

After PTX inspection, this kernel accesses global memory four times, for a measured streaming bandwidth of up to 90 GB/s using *__powf()*, up to 23 GB/s using *powf()* on the Tesla C2050, and 67 GFLOPS and 17 GFLOPS respectively (counting *__powf()* and *powf()* as a single floating point operation). We use a 256 thread block, one per each entry of the *choice_info* data structure, in order to obtain the best performance. These particular values minimize non-coalesced memory accesses and yield high occupancy values.

5.1.2. Tuning our data parallelism approach

For a data parallelism approach, the number of worker ants (threads) per queen ant (blocks) is a degree of freedom studied in Table 3. The 128 thread-block configuration maximizes performance in all benchmark instances, with some of configurations unable to run (denoted n.a) because either the number of worker ants exceeds that of the cities, or the number of cities divided by the number of worker ants is greater than 32 (the maximum number of cities that each worker ant can manage). The tabu list for each queen ant is divided among their worker ants, and placed on a bitwise basis in a single register.

Table 3: Execution times (ms.) on Tesla C2050. We vary the number of worker ants and benchmark instances (n.a. means "not available" due to register constraints).

Worker ants	d198	a280	lin318	pcb442	rat783	pr1002	pr2392
16	10.39	30.06	38.59	101.09	n.a.	n.a.	n.a.
32	6.85	18.68	23.89	62.77	357.24	749.04	n.a.
64	5.06	12.78	15.51	41.17	235.86	474.79	6083.96
128	**4.32**	**11.94**	**15.18**	**38.73**	**207.08**	**391.59**	**5092.27**
256	n.a	15.94	20.89	41.85	245.33	412.20	5680.81
512	n.a	n.a	n.a	n.a	296.90	498.55	6680.39
1024	n.a	n.a	n.a	n.a	n.a	n.a	10037.10

5.1.3. I-Roulette Versus Roulette Wheel

Figure 7 shows the improvement attained by increasing the parallelism on the GPU through our selection method, *I-Roulette*. This method reaches up to 2.36x gain compared to the classic roulette wheel even though it generates many costly random numbers. Roulette wheel compromises GPU parallelism, thus there is a tradeoff between throughput and latency, the former option being favored by a significant margin.

5.1.4. GPU versus CPU

Figure 7 presents execution times on a single-threaded high-end CPU and Tesla C2050 GPU for the set of simulations included within our benchmarking exercise. For a fair comparison, we use hardware platforms of similar cost (between 1500 and 2000 euros). We see that the GPU obtains better performance than its single-threaded CPU counterpart, reaching up to a 21x speed-up factor.

Figure 7 also shows the benefit of having many parallel light-weight threads through a data-parallelism approach, instead of having heavy-weight threads due to the task parallelism approach on GPUs. The task-parallelism can present several operations, including branch statements, that may offer a non-homogeneous computational pattern which is not the ideal framework for GPUs.

For the task parallelism versions, we use 16 CUDA threads with 16 ants running in parallel per thread-block in order to maximize performance. This particular value produces a low GPU resource usage per SM, and it is not well suited for developing high-throughput applications on GPUs. The heavy-weight threads of this design need resources in order to execute their tasks independently, thus avoiding large serialization phases. In CUDA, this is

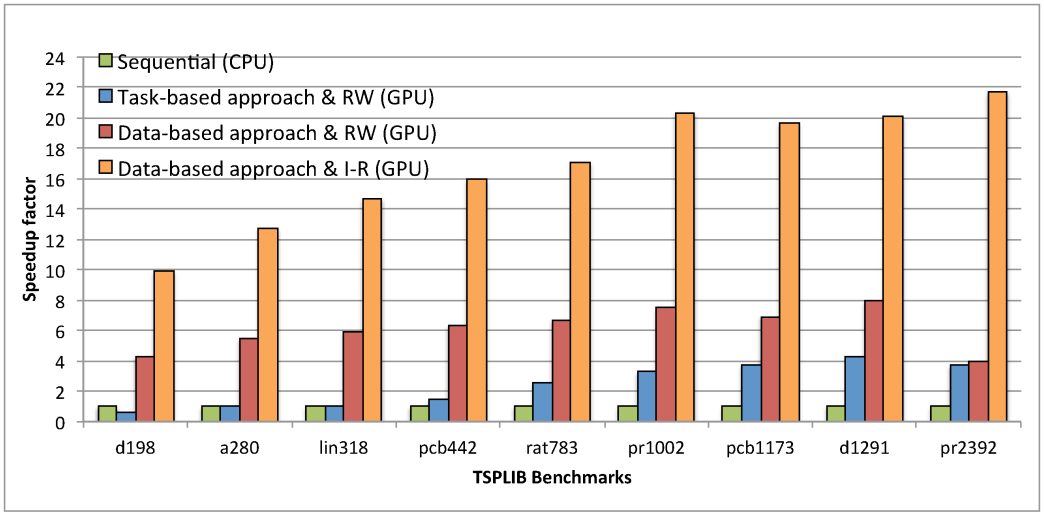

Figure 7: Speed-up factor on different hardware platforms (CPU vs GPU), and enabling different GPU approaches for the tour construction kernel (RW stands for Roulette Wheel, I-R for Independent Roulette).

obtained by distributing those threads among SMs, which is possible by increasing the number of thread-blocks during execution. The task parallelism approach is only rewarded with a maximum of 4.25x gain versus 21.71x reached by the data parallelism alternative, with also worst scalability numbers, although several optimization techniques have been proposed to improve this approach.

5.2. Evaluation of pheromone update kernel

This section discusses performance issues for the pheromone update kernel on Tesla C2050 GPU, and compares them to the CPU-based alternatives.

5.2.1. Evaluation of different GPU algorithmic strategies

The baseline code is our optimal kernel version, which uses atomic instructions and shared memory. This kernel presents the already described tradeoff between the number of accesses to global memory for avoiding costly atomic operations, and the number of those operations (*loads* : *atomic* ratio). The "scatter to gather" computation pattern (version 5) presents a major imbalance between these two parameters, which is reflected in an exponential performance degradation as the problem size grows, as expected (see lower row in Table 4). Tiling (version 4) improves the application bandwidth in the scatter to gather approach. Reduction (version 3) actually reduces the

Table 4: Execution times (ms.) on Tesla C2050 for pheromone update implementations.

Code version	TSPLIB codes (problem size)								
	*d*198	*a*280	*lin*318	*pcb*442	*rat*783	*pr*1002	*pcb*1173	*d*1291	*pr*2392
1. At. Ins. + Tiling	0.18	0.41	0.49	0.54	2.42	3.52	4.68	5.85	18.57
2. Atomic Ins.	0.26	0.45	0.60	0.9	2.49	4.45	5.33	6.01	19.04
3. Ins.& Th. Reduction	25.47	93.93	144.63	516.60	4669.58	12256.4	22651.30	33682.00	390301.32
4. Tiled Sc. to Gather	66.29	211.81	368.91	1321.37	12331.21	32343.64	58740.78	86445.23	1018150.27
5. Scatter to Gather	66.37	260.82	424.11	1534.21	14649.93	39299.14	73384.86	107926.87	1313744.4
Overall Slowdown	368x	636x	865x	2841x	6053x	11164.x	15680x	18448x	70745x

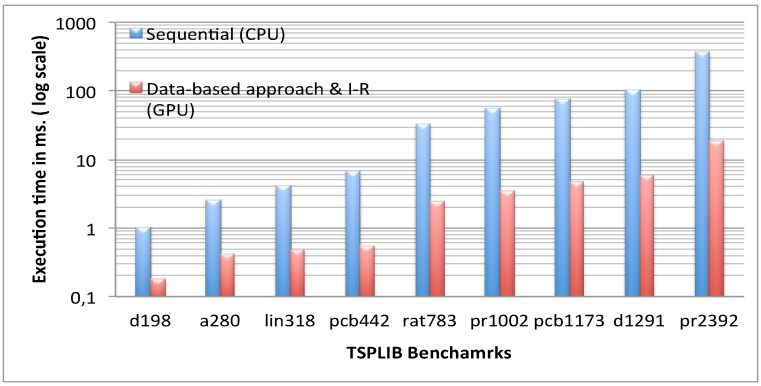

Figure 8: Execution times (ms.) on an Intel Xeon E5620 CPU and a Nvidia Tesla C2050 GPU for the pheromone update stage. We vary the TSPLIB benchmark instance to increase the number of cities (I-R for Independent Roulette).

overall number of accesses to either shared or device memory by halving the number of threads with respect to versions 4 and 5 (and also uses tiling to alleviate device memory use). Even though the number of loads per thread remains the same, the overall number of loads in the *application* is reduced.

5.2.2. GPU versus CPU

Figure 8 shows the execution times (in a log scale) for the best version of the pheromone update kernel compared to the sequential code. The pattern of computation for this kernel is based on data-parallelism, showing a linear speed-up along with the problem size, reaching up to 20x speed-up factor for Tesla C2050.

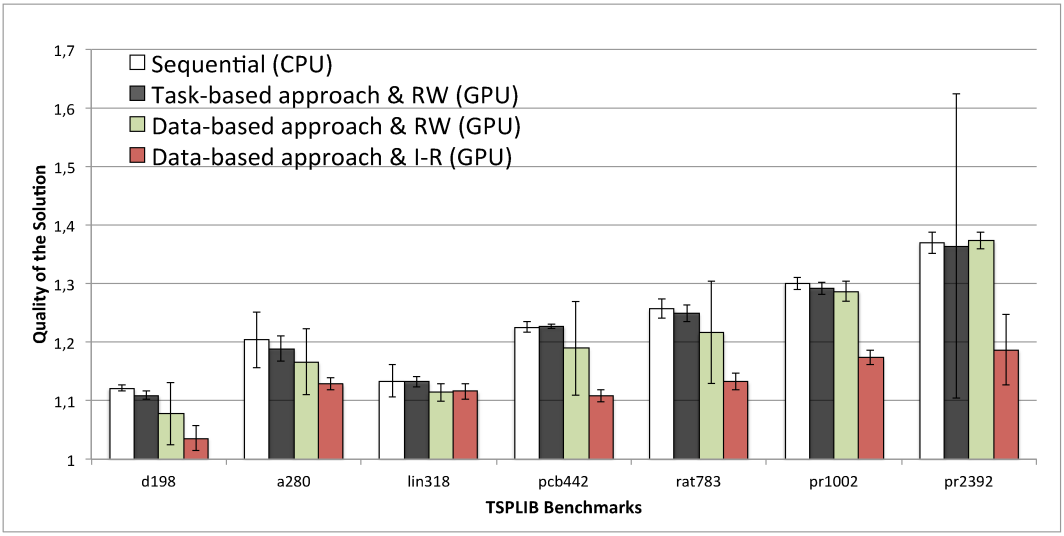

Figure 9: Solution accuracy (averaged over 1000 iterations). A confidence interval of 95% is shown on top of the bars (RW stands for Roulette Wheel, I-R for Independent Roulette).

5.3. Solution Quality

Figure 9 depicts a quality comparison for the solutions we have presented so far. They are normalized with respect to the optimal solution for each case presented in Table 2. We show the result of running all algorithms a fixed number of 1000 iterations and averaged over 5 independent runs, with a 95% confidence interval. Our main conclusion here is that the quality of the tour obtained from GPU codes is similar to that obtained by the sequential code, and sometimes even improves on it.

6. Related Work

6.1. Parallel implementations

Stüzle [8] describes the simplest case of ACO parallelisation, where independent instances of the ACO algorithm are run on different processors. Parallel runs do not incur a communication overhead, and the final solution is chosen from all independent executions. Michel *et al.* [24] present an improved solution based on ant colonies exchanging pheromone information. In more recent work, Chen *et al.* [25] divide the ant *population* into equally-sized sub-colonies, each assigned to a different processor where an optimal local solution is pursued, and information is periodically exchanged among processors. Lin *et al.* [10] decompose the *problem* into subcomponents, with

each subgraph assigned to a different processing unit. To explore a graph and find a complete solution, an ant moves from one processing unit to another, and messages are sent to update pheromone levels. This scheme improves performance by reducing local complexity and memory requirements.

6.2. GPU implementations

6.2.1. Cg

In terms of GPU-specific designs for the ACO algorithm, Jiening *et al.* [26] propose an implementation of the Max-Min Ant System (one of the ACO variants) for the TSP, using C++ and Cg [27]. Attention is focussed on the tour construction stage, and the authors compute the shortest path in the CPU. Catala *et al.* [13] propose two ACO implementations on GPUs, applying them to the Orienteering Problem, using vertex and shader processors.

6.2.2. CUDA

You [28] discusses a CUDA implementation of the Ant System for the TSP. The tour construction stage is identified as a CUDA kernel, being launched by as many threads as ants exists in the simulation. The tabu list for each ant is stored in shared memory, and the pheromone and distances matrices are stored in texture memory. The pheromone update stage is calculated on the CPU. Li *et al.* [14] propose a method based on a fine-grained model for GPU-acceleration, which maps a parallel ACO algorithm to GPU through CUDA. Ants are assigned to processors, which are connected by a population structure [8].

More recently, the TSP has gained some momentum on GPUs and researchers have proposed alternatives methods than ACO, like the construction of the Minimum Spanning Tree (MSP) [29] and the memetic algorithm for VLSI floorplanning [30] with remarkable speedup. Fu *et al.* [12] design the MAX-MIN Ant System for the TSP with MATLAB and the Jacket toolbox for accelerating some parts of the algorithm on GPUs. They highlight the low performance obtained by the traditional *roulette wheel* as a selection process on GPUs, and propose an alternative selection process called "All-In-Roulette", which generates an $m * n$ pseudorandom number matrix, with m being the number of ants and n the number of cities. In his Ph.D. thesis, Weiss applies the ACO algorithm to a data-mining problem [31] and analyzes several ACO designs, highlighting the low GPU performance on previous designs based on task-parallelism.

Although these proposals offer a valid starting point when considering GPU-based parallelisation of ACO, they fail to offer any *systematic* analysis of how close to optimal those solutions are, and they also fail to consider an important component of the ACO algorithm: the pheromone update.

7. Conclusions and Future Work

Ant Colony Optimisation (ACO) belongs to the family of population-based meta-heuristics that has been successfully applied to many NP-complete problems. We have demonstrated that task parallelism used by previous implementation efforts does not fit well on GPU architecture, and, to overcome this issue, an alternative approach based in CUDA and *data parallelism* is provided. This approach enhances the GPU performance by increasing parallelism and avoiding warp divergence, and, combined with an alternative selection procedure that fits better to GPUs, leads to performance gains of more than 20x compared to sequential CPU code executed on a multicore machine.

For the *pheromone update* stage, we are the first to present a complete GPU implementation, including a set of strategies oriented to avoid atomic instructions. We identify potential tradeoffs and investigate several alternatives to sustain gains over 20x in this stage as well. An extensive validation process is also carried out to guarantee the quality of the solution provided by the GPU, and accuracy issues concerning floating-point precision are also investigated.

ACO on GPUs is still at a relatively early stage, and we acknowledge that we have tested a relatively simple variant of the algorithm. But, with many other types of ACO algorithm still to be explored, this field seems to offer a promising and potentially fruitful area of research.

On the hardware side, it is expected to get even higher accelerations on GPUs whenever the problem size keeps growing and larger device memory space is available. Moreover, we may anticipate that the benefits of our approach would also increase when using future GPU generations endowed with thousands of cores and eventually grouped into GPU clusters to lift performance into unprecedented gains where parallelism is called to play a decisive role.

Acknowledgements

This work was partially supported by a travel grant from the EU FP7 NoE HiPEAC IST-217068, the European Network of Excellence on High

Performance and Embedded Architecture and Compilation, by the Spanish MICINN and European Commission FEDER funds under projects Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04, and also by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under projects 00001/CS/2007 and 15290/PI/2010. We also thank NVIDIA for hardware donation under Professor Partnership 2008-2010 and CUDA Teaching Center Award 2011-2012.

References

[1] M. Dorigo, T. Stützle, Ant Colony Optimization, Bradford Company, Scituate, MA, USA, 2004.

[2] M. Dorigo, M. Birattari, T. Stutzle, Ant Colony Optimization, Computational Intelligence Magazine, IEEE 1 (4) (2006) 28 –39.

[3] C. Blum, Ant Colony Optimization: Introduction and recent trends, Physics of Life Reviews 2 (4) (2005) 353–373.

[4] E. Lawler, J. Lenstra, A. Kan, D. Shmoys, The Traveling Salesman Problem, Wiley New York, 1987.

[5] M. Dorigo, A. Colorni, V. Maniezzo, Positive feedback as a search strategy, Tech. Rep. 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy (1991).

[6] M. Dorigo, V. Maniezzo, A. Colorni, The Ant System: Optimization by a colony of cooperating agents, IEEE Transactions on Systems, Man, and Cybernetics-Part B 26 (1996) 29–41.

[7] M. Dorigo, E. Bonabeau, G. Theraulaz, Ant algorithms and stigmergy, Future Gener. Comput. Syst. 16 (2000) 851–871.

[8] T. Stützle, Parallelization strategies for Ant Colony Optimization, in: PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, Springer-Verlag, London, UK, 1998, pp. 722–731.

[9] X. JunYong, H. Xiang, L. CaiYun, C. Zhong, A novel parallel Ant Colony Optimization algorithm with dynamic transition probability, International Forum on Computer Science-Technology and Applications 2 (2009) 191–194.

[10] Y. Lin, H. Cai, J. Xiao, J. Zhang, Pseudo parallel Ant Colony Optimization for continuous functions, International Conference on Natural Computation 4 (2007) 494–500.

[11] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, V. Volkov, Parallel computing experiences with CUDA, IEEE Micro 28 (2008) 13–27.

[12] J. Fu, L. Lei, G. Zhou, A parallel Ant Colony Optimization algorithm with GPU-acceleration based on all-in-roulette selection, in: 2010 Third International Workshop on Advanced Computational Intelligence (IWACI), 2010, pp. 260–264.

[13] A. Catala, J. Jaen, J. Modioli, Strategies for accelerating Ant Colony Optimization algorithms on Graphical Processing Units, in: IEEE Congress on Evolutionary Computation, 2007, pp. 492–500.

[14] 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 (2009) 3707–3716.

[15] Johnson, David S., Mcgeoch, Lyle A., The Traveling Salesman Problem: A Case Study in Local Optimization, John Wiley and Sons, Ltd., 1997, pp. 215–310.

[16] M. Dorigo, Optimization, learning and natural algorithms, Ph.D. thesis, Politecnico di Milano, Italy (1992).

[17] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[18] NVIDIA, NVIDIA CUDA C Programming Guide 3.1.1, 2010.

[19] NVIDIA, NVIDIA CUDA C Best Practices Guide 3.2, 2010.

[20] NVIDIA, NVIDIA CUDA CURAND Library., 2010.

[21] T. Scavo, Scatter-to-gather transformation for scalability (Aug 2010).

[22] G. Reinelt, TSPLIB— A Traveling Salesman Problem library, ORSA Journal on Computing 3 (4) (1991) 376–384.

[23] TSPLIB Webpage, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (February 2011).

[24] R. Michel, M. Middendorf, An island model based Ant System with lookahead for the shortest supersequence problem, in: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, PPSN V, Springer-Verlag, London, UK, 1998, pp. 692–701.

[25] L. Chen, H.-Y. Sun, S. Wang, Parallel implementation of Ant Colony Optimization on MPP, in: Machine Learning and Cybernetics, 2008 International Conference on, Vol. 2, 2008, pp. 981 –986.

[26] W. Jiening, D. Jiankang, Z. Chunfeng, Implementation of Ant Colony Algorithm based on GPU, in: CGIV '09: Proceedings of the 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization, IEEE Computer Society, Washington, DC, USA, 2009, pp. 50–53.

[27] NVIDIA, The Cg Language. Home Page (2008).

[28] Y.-S. You, Parallel ant system for Traveling Salesman Problem on GPUs, in: GECCO 2009 - GPUs for Genetic and Evolutionary Computation., 2009, pp. 1–2.

[29] S. Rostrup, S. Srivastava, K. Singhal, Fast and memory efficient minimum spanning tree on the GPU, in: Proceedings of the 2nd Intl. Workshop on GPUs and Scientific Applications (GPUScA, 2011). Held in conjunction with PACT 2011., Galveston Island, Texas, USA, 2001, pp. 3–13.

[30] S. Potti, S. Pothiraj, GPGPU Implementation of Parallel Memetic Algorithm for VLSI Floorplanning Problem, Springer, 2011, pp. 432–441.

[31] R. M. Weiss, GPU-Accelerated data mining with swarm intelligence, Ph.D. thesis, Department of Computer Science (Macalester College) (2010).