2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

Improving Ant Colony Optimization performance on the GPU using CUDA

Laurence Dawson
School of Engineering and Computing Sciences
Durham University
Durham, United Kingdom
Email: l.j.dawson@dur.ac.uk

Iain Stewart
School of Engineering and Computing Sciences
Durham University
Durham, United Kingdom
Email: i.a.stewart@dur.ac.uk

***Abstract*—We solve the Travelling Salesman Problem (TSP) using a parallel implementation of the Ant System (AS) algorithm for execution on the Graphics Processing Unit (GPU) using NVIDIA CUDA. Extending some recent research, we implement both the tour construction and pheromone update stages of Ant Colony Optimization (ACO) on the GPU using a data parallel approach. In this recent work, roulette wheel selection is used during the tour contruction phase; however, we propose a new parallel implementation of roulette wheel selection called Double-Spin Roulette (DS-Roulette) which significantly reduces the running time of tour construction. We also develop a new implementation of the pheromone update stage. Our results show that compared to its sequential counterpart our new parallel implementation executes up to 82x faster whilst preserving the quality of the tours constructed, and up to 8.5x faster than the best existing parallel GPU implementation.**

I. Introduction

Ant algorithms model the behaviour of real ants to solve a variety of optimization and distributed control problems. Ant Colony Optimization (ACO) [1] is a population-based metaheuristic that has proven to be the most successful ant algorithm for modelling discrete optimization problems. One of the many problems ACO has been applied to is the Travelling Salesman Problem (TSP) in which the goal is to find the shortest tour around a set of cities. Dorigo and Stützle [1] remark that when modelling new algorithmic ideas, the TSP is the standard problem to model due its simplicity; often algorithms that perform well when modelling the TSP will translate with similiar success to other problems.

When modelling the TSP, the simplest implementation of ACO, Ant System (AS), consists of two main stages: *tour construction* and *pheromone update* (an optional additional local search stage may also be applied once the tours have been constructed so as to attempt to improve the quality of the tours before performing the pheromone update stage). The process of tour construction and pheromone update is applied iteratively until a termination condition is met (such as a set number of iterations). By a process known as *stigmergy*, ants are able to communicate indirectly through a *pheromone matrix*. This matrix is updated once each ant has constructed a new tour and will influence successive iterations of the algorithm. As the number of cities to visit increases, so does the computational time required for AS to construct and improve tours. Both the tour construction and pheromone update stages can be performed independently for each ant and this makes ACO particularly suited to GPU parallelization.

As Dorigo and Stützle note [1], there are two main approaches to implementing ACO in parallel: the first maps each ant to an individual processing element (a collection of processing elements therefore constitutes a colony of ants); the second maps an entire colony of ants to a processing element (usually augmented with a method of communicating between the colonies). Multiple colonies are executed in parallel, potentially reducing the number of iterations before termination.

NVIDIA CUDA is a parallel programming architecture for developing general purpose applications for direct execution on the GPU [2]. CUDA exposes the GPU's massively parallel architecture so that parallel code can be written to execute much faster than its optimized sequential counterpart. Although CUDA abstracts the underlying architecture, fully utilising and scheduling the GPU is non-trivial.

This paper improves upon existing parallel ACO implementations on the GPU using NVIDIA CUDA and focuses on both the performance of the algorithm and the quality of the solutions. Using an adapted implementation of Ant System we are able to reduce the computation required and significantly decrease the execution time. Building on the recent work of Cecilia et al. [3] and Delévacq et al. [4] we execute both stages of AS in parallel on the GPU using a data parallel approach. For the tour construction phase, our approach uses a new parallel implementation of the commonly-used roulette wheel selection algorithm (proportionate selection) called DS-Roulette. DS-Roulette exploits the modern hardware architecture, increases parallelism, and decreases the execution time whilst still enabling us to construct high-quality tours. For the pheromone update phase, we adopt the approach taken by $\mathcal{MAX}-\mathcal{MIN}$ Ant System, and allied with a novel implementation we achieve significant speed-ups. We envisage that our main contribution of a new efficient parallel implementation of roulette wheel selection might be more widely applicable within other heuristic problem-solving areas that use proportionate selection such as *Genetic Algorithms* (GA) and *Particle Swarm Optimization* (PSO).

II. Background

In this section we define the TSP and how the TSP can be solved using the AS algorithm. For additional details regarding ACO, various other ant-based algorithms and applications in other domains, we direct the reader to the original works of Dorigo and Stützle [5], [1], [6], [7], [8].

978-1-4799-0454-9/13/$31.00 ©2013 IEEE

Informally, in order to solve the TSP we aim to find the shortest tour around a set of cities. An instance of the problem is a set of cities where for each city we are given the distances from that city to every other city. In more detail, an instance is a set N of cities with an edge (i, j) joining every pair of cities i and j so that this edge is labelled with the distance $d_{i,j}$ between city i and city j. Whilst the aim is to solve the TSP by finding the shortest-length Hamiltonian circuit of the graph (where the length of the circuit is the sum of the weights labelling the edges involved), the fact that solving the TSP is NP-hard means that in practice we can only strive for as good a solution as possible within a feasible amount of time (there is usually a trade-off between these two parameters). Throughout this paper we only ever consider *symmetric* instances of the TSP where $d_{i,j} = d_{j,i}$, for every edges (i, j).

The AS algorithm came after three initially-proposed ant algorithms [1] and consists of two main stages (see Fig. 1): ant solution construction; and pheromone update. These two stages are repeated until a termination condition is met. The colony of ants contains m artificial ants, where m is a user-defined parameter. In the tour construction phase, these m ants construct solutions independently of each other.

```
procedure ACOMetaheuristic
  set parameters, initialize pheromone levels
  while (termination condition not met) do
    construct ants' solutions
    update pheromones
  end
end
```

Fig. 1. Overview of the AS algorithm

To begin, each ant is placed on a randomly chosen start city. The ants then repeatedly apply the random proportional rule, which gives the probability of ant k moving from its current city i to some other city j, in order to construct a tour (the next city to visit is chosen by ant k according to these probabilities). At any point in the tour construction, ant k will already have visited some cities. The set of legitimate cities to which it may visit next is denoted N^k and changes as the tour progresses. Suppose that at some point in time, ant k is at city i and the set of legitimate cities is N^k. The *random proportional rule* for ant k moving from city i to some city $j \in N^k$ is defined via the probability:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \tag{1}$$

where: τ_{il} is the amount of pheromone currently deposited on the edge from city i to city l; η_{il} is a parameter relating to the distance from city i to city l and which is usually set at $1/d_{il}$; and α and β are user-defined parameters to control the influence of τ_{il} and η_{il}, respectively. Dorigo and Stützle [1] suggest the following parameters when using AS: $\alpha = 1$; $2 \leq \beta \leq 5$; and $m = |N|$ (that is, the number of cities), i.e., one ant for each city. We adopt these choices throughout. The probability p_{ij}^k is such that edges with a smaller distance value are favoured.

Once all of the ants have constructed their tours, the pheromone levels of edges must be updated. To avoid stagnation of the population, the pheromone level of every edge is first evaporated according to the user-defined *evaporation rate* ρ (which, as advised by Dorigo and Stützle [1], we take as 0.5). So, each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}. \tag{2}$$

Over time, this allows edges that are seldom selected to be forgotten. Once all edges have had their pheromone levels evaporated, each ant k deposits an amount of pheromone on the edges of their particular tour T^k so that each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k, \tag{3}$$

where the amount of pheromone ant k deposits, that is, $\Delta\tau_{ij}^k$, is defined as:

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

where C^k is the length of ant k's tour T^k. The method of updating the pheromone levels ensures that a shorter tour found by some ant will result in a larger quantity of pheromone being deposited on the edges traversed in this tour. This in turn will increase the chances of one of these edges being selected by some ant (in the next iteration) according to the random proportional rule and becoming an edge in some subsequent tour.

Dorigo and Stützle (see [1]) note that for larger instances of the TSP, the performance of AS decreases. As a result, many alternative algorithms, techniques and extensions have subsequently been proposed including *local search*, *elitist* AS, *rank-based* AS, $\mathcal{MAX} - \mathcal{MIN}$ AS (MMAS), *ant colony system* and the *hyper-cube framework for ACO* [1].

A. CUDA and the GPU

In 2007 NVIDIA introduced CUDA, a parallel architecture designed for executing applications on both the CPU and GPU, along with a new generation of GPU (G80) with dedicated silicon to facilitate future parallel programming [9]. CUDA allows developers to run blocks of code, known as kernels, directly on the GPU using a parallel programming interface and using familiar programming languages (such as C). This broke away from the traditional approach of using complex graphics interfaces such as Cg to harness the power of the GPU for general purpose computation.

B. Blocks and threads

The typical architecture of a CUDA-compatible GPU consists of an array of streaming multiprocessors (SM), each containing a subset of Streaming Processors (SP). When a kernel method is executed, the execution is distributed over a grid of blocks each with their own subset of parallel threads. Each thread within a block is able to communicate with other threads in that block via *shared memory*. Within a block, threads execute in parallel in smaller sub-blocks known as warps, each containing 16-32 threads. There is no guarantee as to the order the warps will execute in; however, within a warp threads can communicate directly with each other using warp level primitives such as *__ballot()*.

C. Memory types

CUDA exposes a set of different memory types to developers, each with unique properties that must be exploited in order to maximize performance. The first type is *register memory*. Registers are the fastest form of storage and each thread within a block has access to a set of fast local registers that exist on-chip. However, each thread can only access it's own registers and as the number of registers is limited per block, blocks with many threads will have fewer registers per thread. For inter-thread communication within a block, shared memory must be used. Shared memory also exists on-chip and is accessible to all threads within the block but is slower than register memory. Newer *Kepler* CUDA GPUs can also communicate directly with other threads within their warp using the new *shfl* method [10]. For inter-block communication and larger data sets, threads have access to *global (DRAM)*, *constant* and *texture memory*. Ever since Fermi, access to global memory is now cached using L1 and L2 caches. Texture and constant memory also benefit from caching, but the initial load will be significantly slower than accessing shared or register memory. When designing a kernel of code for parallel CUDA execution, it is important to fully and properly use the three main memory types (register, shared and global). As Kirk and Hwu note [9], global memory is slow but often large, whereas shared memory is fast but extremely limited (up to 64kB). A common optimization is to load subsets of global memory into shared memory; this approach is known as *tiling*.

III. Related work

As regards parallel implementation, ACO algorithms can be categorized as *fine grained* or *coarse grained*. In a fine grained approach, the ants are individually mapped to processing elements with communication between processing elements being ant to ant. In a coarse grained approach, entire colonies are mapped to processing elements with communication between colony to colony [1] (adopting a coarse grained approach with no communication between colonies is equivalent to executing the algorithm multiple times on the same instance but independently). In this section we will review existing parallel ACO contributions that target the GPU in more detail.

Catala et al. [11] presented one of the first GPU implementations of ACO targeted at modelling the *Orienteering Problem*. At the time of publication, CUDA had not yet been released and so their implementation relies upon a previous direct GPU interface using graphics paradigms to solve general-purpose problems. Jiening et al. [12] implemented the MMAS algorithm to solve the TSP. This early work was also published prior to the release of CUDA and the authors note that their implementation was much more complex than its CPU counterpart. Their implementation of a parallel tour construction phase resulted in a small but incremental speedup. Using the *Jacket* toolbox for *MATLAB*, Fu et al. [13] implemented a parallel version of MMAS for the GPU to solve the TSP. Their approach focused more on a MATLAB implementation and less on the GPU (as parallelization is handled by Jacket). They reported a speedup; however, their comparative CPU implementation was MATLAB-based which is inherently slower due to being an interpreted language.

Zhu and Curry [14] modeled an ACO Pattern Search algorithm for nonlinear function optimization problems using CUDA. They reported speedups of around 250x over the sequential implementation. Bai et al. [15] implemented a coarse-grained multiple colony version of MMAS using CUDA to solve the TSP. Each ant colony is mapped to a thread block and within the block each thread is mapped to an ant. Their approach yields tours with a quality comparable to the CPU implementation but the speedup reported is only around 2x. You [16] presented an implementation of the AS algorithm using CUDA to solve the TSP. Each thread is mapped to an ant but each thread block is part of a larger colony. The speedup reported for this simpler approach is around 2-20x when the number of ants is equal to the number of cities.

Weiss [17] developed a parallel version of AntMinerGPU (an extension of the MMAS algorithm). As in [15] and [16], each ant within the colony is mapped to an individual CUDA thread. Weiss argues that this approach, coupled with the AntMinerGPU algorithm, allows for larger population sizes to be considered. Weiss moves all stages of the algorithm to the GPU to avoid costly transfers to and from the GPU, a practice that is advocated by the programming guide [2]. Weiss tested the implementation against two problems from the UCI Machine Learning Repository: Wisconsin Breast Cancer; and Tic-Tac-Toe. The GPU version on AntMiner was up to 100x faster than the CPU implementation on large population sizes. Finally, Weiss noted that the implementation could easily be extended to support multiple colonies across multiple GPUs with possible communication.

A. Data-parallelism

As Cecilia et al. note [3], the majority of the current contributions fail to implement the entire ACO algorithm on the GPU and provide no systematic analysis of how best to implement the algorithms in parallel. All of the above implementations also adopt a task-based parallelism strategy that maps ants directly to threads. The following two papers adopted a novel data parallel approach that maps ants to thread blocks so as to better utilise the GPU architecture.

Cecilia et al. [3] implemented the AS algorithm for solving the TSP on the GPU using CUDA. They note that the existing task-based approach of mapping one ant per thread is fundamentally not suited to the GPU. With a task-based approach, each thread must store each ant's memory (e.g., list of visited cities). This approach works for small tours but quickly becomes problematic with larger tours, as there is limited shared memory available. The alternatives are to use fewer threads per block, which reduces GPU occupancy, or to use global memory to store each ant's memory, which dramatically reduces the kernel's performance.

The second issue with task-based parallelism (as discussed in [3]) is warp-branching. As ants construct a tour, their execution paths generally differ due to conditional statements inherent when using roulette wheel selection on the output of the random proportional rule (we will discuss this further in Section 4). When a warp branches, all threads within the branch are serialized and execute sequentially until the branching section is complete, thus significantly impeding the performance of branching code. Cecilia et al. present data-parallelism as an alternative to task-based parallelism so as to avoid these issues and increase performance.

Data parallelism avoids warp-divergence and memory issues by mapping each ant to a thread block. All threads within the thread block then work in cooperation to perform a common task such as tour construction. Cecilia et al. use a fixed-size thread block that tiles to match the size of the problem. A thread is responsible for an individual city and the probability of visiting a city can be calculated without branching the warp by using a proportionate selection method known as I-Roulette [3]. Cecilia et al. also implement the pheromone update stage on the GPU and provide two alternative ways to perform this but conclude that the simpler method of using atomic methods is significantly faster and will likely to continue to be so due to ongoing hardware improvements by NVIDIA. A speedup factor of up to 20x is reported when both the tour construction and pheromone update phases are executed on the GPU (with the majority of the execution time spent on the tour construction phase).

The second data-parallel paper was presented by Delèvacq et al. [4] and implemented the MMAS algorithm with 3-opt local search for solving the TSP using CUDA. Delevacq et al. conduct a similar survey of data-parallelism against task-based parallelism and conclude that a task-based approach is more suitable due to the aforementioned reasoning and include detailed benchmarks to support this assertion.

To improve solution quality in [4], after a block constructs a new ant tour, the 3-opt local search algorithm is applied. This process involves removing three edges from the tour and rearranging them in such a way that the resulting tour has a smaller length than the initial tour, if this is at all possible. This is a complex and time consuming process but can yield near optimal results. Delèvacq et al. note that the data structures required to compute the 3-opt search must be stored in global memory, therefore the execution is slow.

Building on their initial offering, Delvacq et al. also implemented multiple colonies to run in parallel across separate GPUs. The same test instances of the TSP were considered (as were considered for the implementations in [3] mentioned above) and the results showed up to a 20x speedup against a sequential implementation of the MMAS algorithm with local search disabled. With 3-opt local search enabled, the quality of the tours constructed improved significantly; however, the speedup was notably reduced.

IV. Implementation

In this section we present our parallel implementation of the AS algorithm for execution on the GPU. We base our parallelisation strategy on the works of Cecilia et al. [3] and Delvacq et al. [4] and adopt a data-parallel approach, mapping each ant in the colony to a thread block. As discussed above, we execute each stage of the algorithm (see Fig. 1) on the GPU to maximize performance.

The first stage of the algorithm parses the city data and allocates memory and the relevant data structures. For any given city set of size n, the city-to-city distances are loaded into an $n \times n$ matrix (recall, $d_{i,j} = d_{j,i}$, for every pair of distinct cities i and j). Ant memory is allocated to store each ant's current tour and tour length. A *pheromone matrix* is initialized on the GPU to store pheromone levels and a secondary structure called *choice_info* is used to store the product of the denominator of equation 1 (an established optimization, detailed in [1], as these values do not change during each iteration of the algorithm). Once the initialization is complete, the pheromone matrix is artificially seeded with a tour generated using a greedy search as recommended in [1] (other approaches can also be used such as seeding large values to encourage search space exploration).

A. Tour construction

In Section 2 we gave a broad overview of the tour construction phase that is applied iteratively until a new tour is constructed. Dorigo and Stützle provide a detailed description of how to implement tour construction sequentially along with pseudo-code [1] which we summarize in Fig. 2 (where the number of cities is n with the cities named as $1, 2, \ldots, n$).

```
procedure ConstructSolutions
  tour[1] ← place the ant on a random initial city
  for j = 2 to n − 1 do
    for l = 1 to n do
      probability[l] ← CalcProb(tour[1 . . . j − 1],l)
    end-for
    tour[j] ← RouletteWheelSelection(probability)
  end-for
  tour[n] ← remaining city
  tour_cost ← CalcTourCost(tour)
end
```

Fig. 2. Overview of an ant's tour construction

After the initial random city is chosen, the first inner for-loop repeats $n-2$ times to build an complete tour (note that there are only $n-2$ choices to make as once $n-2$ cities have been chosen there is no choice as regards the last). Within the inner for-loop, the probability of moving from the last visited city to all other possible cities is calculated. Calculating the probability consists of two stages: retrieving the value of *choice_info*$[j][l]$ (the numerator in equation 1) and checking if city l has already been visited in the current iteration (in which case the probability is set to 0). The next city to visit is selected using roulette wheel selection.

Roulette wheel selection is a prime example of something that is sequentially trivial yet requires additional consideration when implemented using a GPU. *Roulette wheel selection* is illustrated in Table. I where 1 item has to be chosen from 5 in proportion to the value in the first column labelled 'input'. The first step is to reduce the set of input values so as to obtain cumulative totals (as is depicted in the second column labelled 'reduced'). The reduced values are normalized so that the sum of all input values normalizes to 1 (see the third column labelled 'normalized') and the portion of the roulette wheel corresponding to some item is calculated (see the fourth column labelled 'range'). The final step is to generate a random number that is greater than 0.0 and at most 1.0, and then to use the ranges so as to choose an item (so if 0.5 was generated, for example, then we would choose item 5, as 0.5 is greater than 0.365 but less than or equal to 0.875). Sequentially this process is trivial; however, the linear nature of the algorithm, the divergence in control flow, parallel random number generation and the need for constant thread synchronization means we have to work harder in our GPU setting.

TABLE I. ROULETTE WHEEL SELECTION

input	reduced	normalized	range
0.1	0.1	0.1	> 0.0 & ≤ 0.1
0.3	0.4	0.25	> 0.1 & ≤ 0.25
0.2	0.6	0.375	> 0.25 & ≤ 0.375
0.8	1.4	0.875	> 0.375 & ≤ 0.875
0.2	1.6	1.00	> 0.875 & ≤ 1.0

Cecilia et al. [3] address these issues by implementing *Independent-Roulette* (I-Roulette), a parallel implementation of roulette wheel selection. In I-Roulette, each thread l calculates the probability of moving to city j. Each thread then checks to see whether the city l has been previously visited and multiplies the probability just calculated by a random number. Finally, the n resulting values undergo a reduction and the largest value is selected as the next city (by a reduction we mean a process by which the values are pairwise linearly compared with the largest being retained). There are some issues with this approach.

- The reduction is performed on the entire thread block and this prohibits synchronization between warps so that idle warps result.
- The random numbers generated can dramatically decrease the influence of the heuristic and the pheromone information. This in turns leads to a reduction in the quality of the tours generated.
- A random number must be generated for each thread for $n-2$ iterations in order to build a complete tour.
- From the source code provided by Cecilia et al., it can be observed that often a single thread is used for various stages of the algorithm resulting in warp divergence and reduced parallelism.
- The entire tour is stored in shared memory.
- When constructing tours for large instances, shared memory is often exhausted.

To address these issues we present *Double-Spin Roulette* (DS-Roulette), a highly parallel roulette selection algorithm that exploits warp-level parallelism, reduces shared memory dependencies, and reduces the overall instruction count and work performed by the GPU. In the sequential implementation of roulette wheel selection, each ant constructs a tour one city at a time and each ant is processed consecutively. The first level of parallelism we employ is to execute the tour construction stage in parallel. Using a data-parallel approach (as suggested in [3] and [4]), each ant is mapped to a thread block (so that m ants occupy m blocks).

DS-Roulette consists of three main stages that are executed in succession (see Fig. 3). Thread synchronization is utilised after each stage to ensure that all threads within the block have finished the previous stage before proceeding. Cecilia et al. note that due to having a fixed maximum thread count and limited shared memory available, tiling threads across the block to match the number of cities yielded the best results. Building upon this observation, the first stage of DS-Roulette tiles 4 thread warps (128 threads) so as to provide complete coverage of all potential cities (again illustrated in Fig. 3). We will henceforth refer to a tiled warp consisting of 32 threads as a sub-block that represents a block of possible cities.

In the first stage, each thread within the sub-block checks if the city it represents has previously been visited in the current tour. This valued is stored in shared memory and is known as the tabu value. A warp-level poll is then executed to determine if any valid cities remain in the sub-block (this reduces redundant memory accesses). If valid cities remain then each thread retrieves the respective probability from the *choice_info* array and multiplies the probability by the associated *tabu value*. The sub-block then performs a warp-reduction (see Fig. 4) on the probabilities to calculate the sub-block probability and this is stored in shared memory. Warp-level parallelism implicitly guarantees that all threads execute without divergence throughout the stage and this significantly decreases the execution time. Once the probability for the subblock is calculated, the sub-blocks then tile to ensure complete coverage of all cities within the tour.

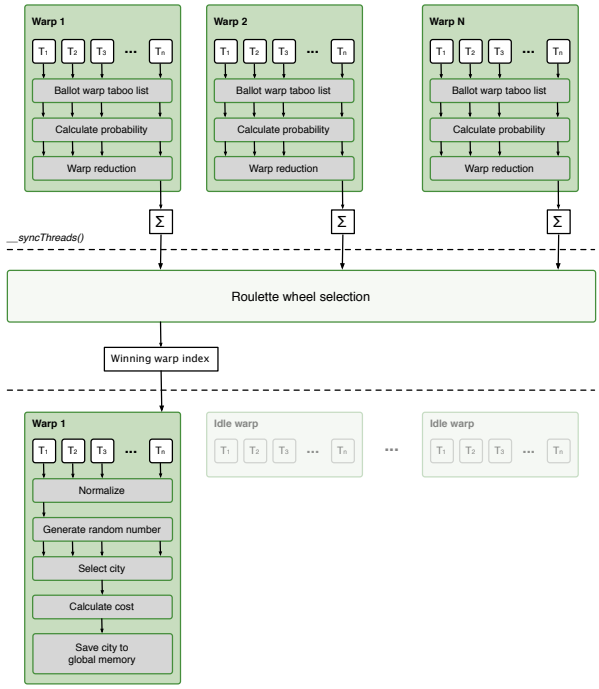

Fig. 3. An overview of the double-spin roulette algorithm which is run by each ant

```
__device__ void
warp_reduce (int tid, float in, float
*data){
  int idx = (2 * tid - (tid & 31) );
  data[idx] = 0;
  idx += 32;
  float t = data[idx] = in;

  data[idx] = t = t + data[idx - 1 ];
  data[idx] = t = t + data[idx - 2 ];
  data[idx] = t = t + data[idx - 4 ];
  data[idx] = t = t + data[idx - 8 ];
  data[idx] = t = t + data[idx - 16 ];
}
```

Fig. 4. The warp-reduce method [18]

At the end of stage one, the results are a list of sub-block probabilities. Stage 2 performs roulette-wheel selection on this set of probabilities to chose a specific sub-block from which the next city to be visited will be chosen. If the tour size were 256 cities, for example, there would be 8 potential sub-blocks previously calculated by 4 warps. This is the first spin of the roulette wheel (other methods such as greedy selection can also be utilized here). Once a warp has been chosen, the value is then saved to shared memory.

In the final stage of DS-Roulette, we limit the execution to the chosen sub-block. Using the block value calculated in stage 2, the first 32 threads load, from shared memory, the probabilities calculated by the winning sub-block in stage 1. Each thread then loads the total probability of the sub-block and the probabilities are normalized. As each thread is accessing the same 32-bit value from shared memory, the value is broadcast to each thread eliminating bank conflicts and serialization. A single random number is then generated and each thread checks if the number is within their range thus completing the second spin of the roulette wheel. The winning thread then saves the next city to shared memory and global memory, and updates the total tour cost. After $n-2$ iterations, the tour cost is saved to global memory using an atomic *max* operator. This value is subsequently used during the pheromone update stage.

Using this technique we address the issues raised by I-Roulette. Instead of reducing and normalizing the entire thread block, we split the block into smaller sub-blocks from which we approximate where the next city is to be selected from. Roulette wheel selection is applied twice to calculate the next city. This process drastically reduces the synchronisation overhead and the need for each thread to generate a costly random number. We exploit warp-level scheduling to avoid additional computation and this reduce the total instruction count. DS-Roulette preserves the influence of both the heuristic and the pheromone information and leads to higher quality tours. At any point, only the last visited city is required to be kept in shared memory thus reducing the total shared memory required. This in turn increases the occupancy of the GPU when larger tour instances are used.

B. Pheromone update

The last stage of the AS algorithm is pheromone update which consists of two stages: pheromone evaporation; and pheromone deposit. The pheromone evaporation stage (see equation 3) is trivial to parallelize as all edges are evaporated by a constant factor ρ. A single thread block is launched which maps each thread to an edge and reduces the value using ρ. A tiling strategy is once again used to cover all edges (an alternative strategy was originally used to map each row of the pheromone matrix to a tiling thread block; however, this was found to be considerably slower in practice).

The second stage, pheromone deposit (see equation 4), deposits a quantity of pheromone for each edge belonging to a constructed tour for each ant. Cecilia et al. [3] note that as each ant will perform this stage in parallel, atomic operations must be used to ensure correctness of the pheromone matrix. Atomic operations are expensive and Cecilia et al. provide an alternative approach using *scatter to gather transformations*. Using this approach removes the dependency on atomic operations; however, it results in far more global memory loads which severely impedes the performance. As a result they conclude that although there is a dependency on atomic operations, the implementation is faster than other alternatives.

Other ant algorithms such as elitist ant, MMAS and ACO apply pre-conditions on the pheromone update state. For example, when using MMAS only the iteration's best ant deposits pheromone. To reduce the usage of atomic operations and increase convergence speed, we adopt the pheromone update stage from MMAS into our AS implementation. At the end of the tour construction stage, each ant performs a single atomic *min* operation on a memory value storing the tour length. This single operation per block allows the lowest tour value to be saved without additional kernels and is inexpensive. For the tour construction stage we then launch m thread blocks (representing m ants) which then individually check if their tour cost is equal to the lowest overall cost and if so deposit pheromone without the need for atomics.

V. Results

In this section, we present the results obtained by executing our implementation on various instances of the TSP and we compare these results to other parallel and sequential implementations. We use the previously mentioned standard ACO parameters but modify the value of ρ from 0.5 to 0.1 to reduce the rate of evaporation on the pheromone matrix (reducing the rate of evaporation will preserve edges within the pheromone matrix for longer) for both GPU and CPU implementations. As we have modified the pheromone deposit stage to only use the best ant, significantly less pheromone is deposited at the end of each iteration. The reduced evaporation rate reflects this change and ensures that the pheromone matrix still has sufficient pheromone to influence the tour construction.

For testing our implementation we use an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield). The GPU contains 580 CUDA cores and has a processor speed of 1544 MHz. As the card is from the Fermi generation, it uses 32 threads per warp and up to 1024 threads per thread block with a maximum shared memory size of 64 Kb. The CPU has 4 cores which support up to 8 threads with a clock speed of 3.06 GHz. Our implementation was written and compiled using the latest CUDA tooklkit (v5.0) for C and executed under Ubuntu 12.10. To match the setup of Cecilia et al. [3], timing results are averaged over 100 iterations of the algorithm with 10 independant runs.

A. Solution Quality

To evaluate the quality of the tours constructed, we compared the results of our new GPU implementation against an existing CPU implementation of AS for the set number of iterations. Our new approach was able to match and in most cases reduce the length of the tours constructed when using indentical parameters and number of iterations. As is expected when using the AS algorithm, the results are not optimal and the quality of tours constructed decreases as the number of cities increases. However, this was seen across both versions and can be improved by implementing local search. Fig.5 shows a comparison of the average quality of tours obtained via the existing CPU and new GPU implementation.

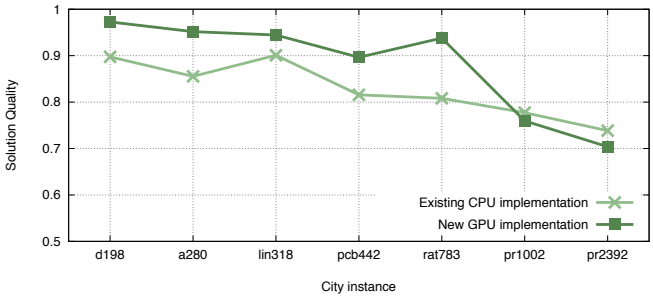

Fig. 5. Quality of existing CPU and new GPU tour construction

TABLE II. AVERAGE TOUR LENGTHS OBTAINED

Instance	Optimal	CPU	GPU
$d198$	15780	17583	16222
$a280$	2579	3015	2710
$lin318$	42029	46651	44495
$pcb442$	50778	62255	56639
$rat783$	8806	10896	9390
$pr1002$	259045	333262	341080
$pr2392$	378032	511977	537127

B. Benchmarks

In Table. III we present the execution times for various instances of the TSP and in Fig. 6 and Fig. 7 we present the speedup attained against the standard CPU implementation and the best existing GPU implementation (using the same configuration as our GPU implementation). In Table. IV we break down the time spent on each stage of the algorithm for our GPU implementation. For the sequential implementation, we compare against ACOTSP (source available at [19]) which is the standard sequential implementation used (see, e.g., [3], [4]). We also compare against the best current parallel GPU implementation of AS provided by Cecilia et al. [3]. The same instances of the TSP were repeated from [3]. The results for the existing CPU and GPU implementations were obtained by repeating their executions on our hardware to ensure that the comparison is fair and that any speedup is not a product of hardware or software differences.

TABLE III. AVERAGE EXECUTION TIMES (MS)

Instance	CPU	Amos GPU	Our GPU	Speedup CPU	Speedup GPU
$d198$	48.37	7.60	1.16	41.79x	6.56x
$a280$	123.13	18.61	2.68	45.86x	6.93x
$lin318$	175.57	26.43	3.39	51.79x	7.79x
$pcb442$	482.19	66.34	7.79	61.86x	8.51x
$rat783$	3059.70	332.15	42.70	71.65x	7.78x
$pr1002$	7004.72	646.90	85.11	82.30x	7.60x
$nrw1379$	17711.68	1687.63	323.00	54.83x	5.22x
$pr2392$	97850.65	8026.75	1979.31	49.44x	4.06x

TABLE IV. EXECUTION TIMES FOR EACH STAGE OF OUR PROPOSED GPU IMPLEMENTATION (MS)

Instance	Tour Construction	Pheromone Update
$d198$	1.10	0.062
$a280$	2.63	0.059
$lin318$	3.30	0.087
$pcb442$	7.72	0.079
$rat783$	42.56	0.138
$pr1002$	84.96	0.153
$nrw1379$	322.71	0.296
$pr2392$	1977.93	1.380

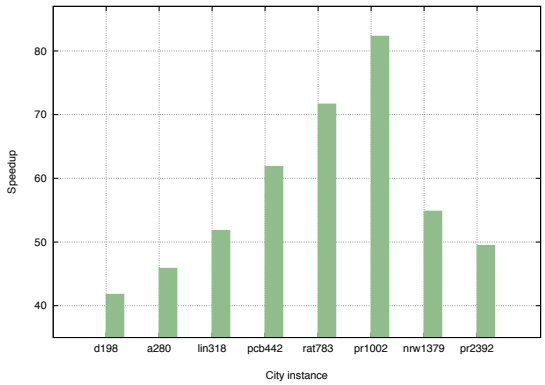

Fig. 6. Speedup of execution against the standard CPU implementation

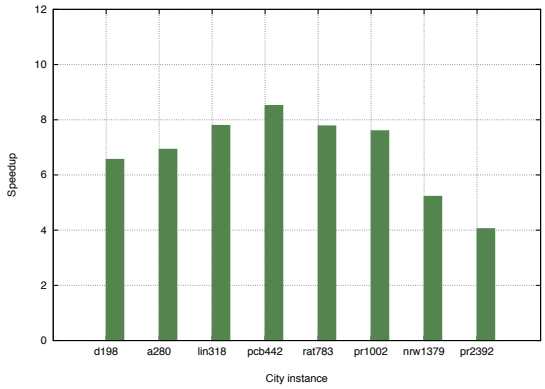

Fig. 7. Speedup of execution against the best existing GPU implementation

The results show a speedup of up to 82x faster than the sequential implementation and up to 8.5x faster than the current best parallel implementation. The tour construction stage takes the majority of the total execution time and varies between 4-8.5x faster than the existing GPU implementation. As previously mentioned, the tour construction stage uses a new efficient implementation of roulette wheel selection and we believe our implementation will be able to bring similar speedups to other algorithms limited by the execution time of proportionate selection. The pheromone update implementation is between 1-9x faster than the existing GPU implementation. As the performance of atomic operations is further optimised with subsequent hardware releases from NVIDIA, the execution time of the pheromone update stage will decrease further without additional software optimization.

The size of the shared memory available for each thread block can be reduced (which in turn increases the L1 cache size available) by altering the *preferred cache configuration*. By reducing the shared memory available we observe that for larger tour sizes, the reduction in shared memory size no longer has any affect. From this we can infer that the shared memory in larger instances is exhausted which forces threads to use additional slower global memory and reduces the efficiency of the tour construction phase. The warp-reduce method uses double the shared memory required over a simpler branching implementation and we believe that this is the likely cause of the decreased speedup observed for larger TSP instances (see Fig. 8). Due to this memory limitation, we did not test any TSP instances larger than 2392 cities.

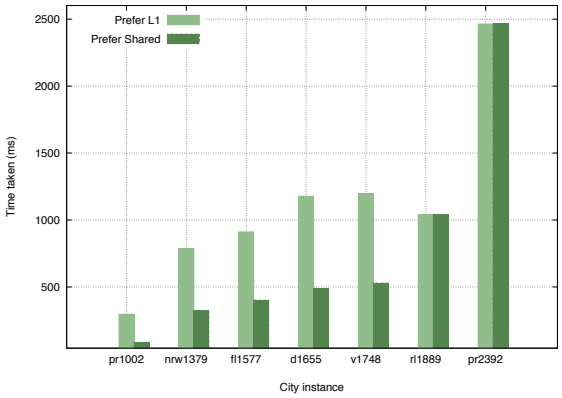

Fig. 8. Observed execution speeds as a product of varying the size of the L1 cache

VI. Conclusions

In this paper we have presented a new data-parallel GPU implementation of the AS algorithm that executes both the tour construction and pheromone update stages on the GPU. Our results show a speedup of up to 8.5x faster than the best existing GPU implementation and up to 82x faster than the sequential counterpart. For larger data sets we observed that shared memory usage can often be exhausted causing the performance of the algorithm to degrade. Our implementation is able to match the quality of solutions generated sequentially. However, we can still improve things further. Our future work will aim to reduce the shared memory requirements of DS-Roulette and to augment our implementation with 2-opt local search. Our primary contribution of an efficient parallel implementation of roulette wheel selection contributed significantly to the reported speedups and we envisage the parallel algorithm might be more widely applicable within other heuristic problem-solving areas.

Acknowledgment

The authors would like to thank José Cecilia for providing the source code of the implementation in [3]. This allowed us to identify areas to improve upon, resulting in the development of DS-Roulette.

References

[1] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[2] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (last accessed 29/01/2013).

[3] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldon, "Enhancing data parallelism for ant colony optimization on GPUs," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.

[4] A. Delèvacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 52–61, 2013.

[5] M. Dorigo, "Optimization, learning and natural algorithms," Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

[6] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," in *Fifth Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS)*, ser. Lecture Notes in Computer Science, M. Dorigo, L. M. Gambardella, M. Birattari, A. Martinoli, R. Poli, and T. Stützle, Eds., vol. 4150. Springer Verlag, 2006, pp. 224–234.

[7] T. Stützle, "Parallelization strategies for ant colony optimization," in *Fifth Int. Conf. on Parallel Problem Solving from Nature (PPSN-V)*. Springer-Verlag, 1998, pp. 722–731.

[8] T. Stützle and H. H. Hoos, "MAX-MIN ant system," *Future Gener. Comput. Syst.*, vol. 16, no. 9, pp. 889–914, Jun. 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=348599.348603

[9] D. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[10] NVIDIA, "Inside Kepler," http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0642-GTC2012-Inside-Kepler.pdf (last accessed 29/01/2013).

[11] A. Catala, J. Jaen, and J. Modioli, "Strategies for accelerating ant colony optimization algorithms on graphical processing units," in *IEEE Congress on Evolutionary Computation (CEC)*, Sept. 2007, pp. 492–500.

[12] W. Jiening, D. Jiankang, and Z. Chunfeng, "Implementation of ant colony algorithm based on GPU," in *Sixth Int. Conf. on Computer Graphics, Imaging and Visualization (CGIV)*, Aug. 2009, pp. 50–53.

[13] J. Fu, L. Lei, and G. Zhou, "A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection," in *Third Int. Workshop on Advanced Computational Intelligence (IWACI)*, Aug. 2010, pp. 260–264.

[14] W. Zhu and J. Curry, "Parallel ant colony for nonlinear function optimization with graphics hardware acceleration," in *Proc. Int. Conf. on Systems, Man and Cybernetics*, Oct. 2009, pp. 1803–1808.

[15] H. Bai, D. Ouyang, X. Li, L. He, and H. Yu, "MAX-MIN ant system on GPU with CUDA," in *Fourth Int. Conf. on Innovative Computing, Information and Control (ICICIC)*, Dec. 2009, pp. 801–804.

[16] Y. You, "Parallel ant system for traveling salesman problem on GPUs," GPUs for Genetic and Evolutionary Computation, GECCO, http://www.gpgpgpu.com/gecco2009 (last accessed 29/01/2013).

[17] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[18] W.-M. W. Hwu, *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2011.

[19] M. Dorigo, "Ant Colony Optimization - Public Software," http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html (last accessed: 29/01/2013.