Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU

Laurence Dawson and Iain A. Stewart

School of Engineering and Computing Sciences,
Durham University, Durham, United Kingdom
{l.j.dawson,i.a.stewart}@durham.ac.uk

Abstract. For solving large instances of the Travelling Salesman Problem (TSP), the use of a candidate set (or candidate list) is essential to limit the search space and reduce the overall execution time when using heuristic search methods such as Ant Colony Optimisation (ACO). Recent contributions have implemented ACO in parallel on the Graphics Processing Unit (GPU) using NVIDIA CUDA but struggle to maintain speedups against sequential implementations using candidate sets. In this paper we present three candidate set parallelization strategies for solving the TSP using ACO on the GPU. Extending our past contribution, we implement both the tour construction and pheromone update stages of ACO using a data parallel approach. The results show that against their sequential counterparts, our parallel implementations achieve speedups of up to 18x whilst preserving tour quality.

Keywords: Ant Colony Optimization, Graphics Processing Unit, CUDA, Travelling Salesman.

1 Introduction

Ant algorithms model the behaviour of real ants to solve a variety of optimization and distributed control problems. Ant Colony Optimization (ACO) [7] is a population-based metaheuristic that has proven to be the most successful ant algorithm for modelling discrete optimization problems. One of these problems is the Travelling Salesman Problem (TSP) in which the goal is to find the shortest tour around a set of cities. Dorigo and Stützle note [7] that the TSP is often the standard problem to model as algorithms that perform well when modelling the TSP will translate with similiar success to other problems. Dorigo and Stützle also remark [7] that ACO can be applied to the TSP easily as the problem can be directly mapped to ACO. For this reason, solving the TSP using ACO has attracted significant research effort and many approaches have been proposed.

The simplest of these approaches is known as Ant System (AS) and 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 successively until a termination condition is met (such as a set number of iterations or minimum solution quality is attained). Through a process known as *stigmergy*, ants are able to communicate indirectly through *pheromone trails*. These trails are 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 effort and thus time required for AS to construct and improve tours. The search effort can be reduced through use of a *candidate set* (or *candidate list*). In the case of the TSP a candidate set provides a list of nearest cities for each city to visit. During the tour construction phase these closest cities will first be considered and only when the list has been exhausted will visiting other cities be permitted.

As both the tour construction and pheromone update stages can be performed independently for each ant in the colony and this makes ACO particularly suited to parallelization. There are two main approaches to implementing ACO in parallel which are known as fine and coarse grained. The fine grained approach maps each ant to an individual processing element. The coarse grained approach maps an entire colony to a processing element [7].

NVIDIA CUDA is a parallel programming architecture for developing general purpose applications for direct execution on the GPU [8] for potential speed increases. Although CUDA abstracts the underlying architecture of the GPU, fully utilising and scheduling the GPU is non-trivial.

This paper builds upon our past improvements to existing parallel ACO implementations on the GPU using NVIDIA CUDA [3]. We observed that parallel implementations of ACO on the GPU fail to maintain their speedup against their sequential counterparts that use candidate sets. This paper addresses this problem and explores three candidate set parallelization strategies for execution on the GPU. The first adopts a naive ant-to-thread mapping to examine if the use of a candidate set can increase the performance; this naive approach (in the absence of candidate sets) has previously been shown to perform poorly [2]. The second approach extends our previous data parallel approach (as pioneered by Cecilia et al. [2] and Delévacq et al. [4]), mapping each ant to a thread block. Through the use of warp level primitives we manipulate the block execution to first restrict the search to the candidate set and then expand to all available cities dynamically and without excessive thread serialization. Our third approach also uses a data parallel mapping but compresses the list of potential cities outside of the candidate set in an attempt to further decrease execution time.

We find that our data parallel GPU candidate set mappings reduce the computation required and significantly decrease the execution time against the sequential counterpart when using candidate sets. By adopting a data parallel approach we are able to achieve speedups of up to 18x faster than the CPU implementation whilst preserving tour quality and show that candidate sets can be used efficiently in parallel on the GPU. As candidate sets are not unique to ACO, we predict that our parallel mappings may also be appropriate for other heuristic problem-solving algorithms such as *Genetic Algorithms*.

2 Background

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. Throughout this paper we only ever consider *symmetric* instances of the TSP where $d_{i,j} = d_{j,i}$, for every edge (i, j). For a more detail on the TSP we direct readers to [3].

The AS algorithm consists of two main stages: ant solution construction; and pheromone update [7] and are repeated until a termination condition is met. 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 [7] 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. 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 level of every edge is evaporated according to the user-defined *evaporation rate* ρ (which, as advised by Dorigo and Stützle [7], we take as 0.5). So, each pheromone level τ_{ij} becomes:

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

This allows edges that are seldom selected to be forgotten. After evaporation, 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. Depositing pheromone increases the chances of one of the shorter edges being selected by an ant in a subsequent tour.

2.1 Candidate Sets

For larger instances of the TSP, the computational time required for the tour construction phase of the algorithm increases significantly. A common solution to this problem is to limit the number of available cities which we refer to as a candidate set. In the tour construction phase the ant will first consider all closely neighbouring cities. If one or more of the cities in the candidate set has not yet been visited, the ant will apply proportional selection on the closely neighbouring cities to determine which city to visit next. If no valid cities remain in the candidate set, the ant then applies an arbitrary selection technique to pick from the remaining unvisited cities. Dorigo and Stützle [7] utilise greedy selection to pick a city with the highest pheromone value. Randall and Montgomery [10] propose several new dynamic candidate set strategies, however, for this paper we will only focus on static candidate sets.

2.2 CUDA and the GPU

NVIDIA CUDA is a parallel architecture designed for executing applications on both the CPU and GPU. CUDA allows developers to run blocks of code, known as kernels, on the GPU for potential speed increases. A CUDA 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.

CUDA exposes a set of memory types each with unique properties that must be exploited in order to maximize performance. The first type 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. 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. For inter-block communication and larger data sets, threads have access to *global*, *constant* and *texture memory.*

3 Related Work

In this section we will briefly cover our past parallel ACO contribution and detail a new parallel ACO implementation. For a comprehensive review of all ACO GPU literature to date we direct readers to [3].

In our previous contribution [3] we presented a highly parallel GPU implementation of ACO for solving the TSP using CUDA. By extending the work of Cecilia et al. [2] and Delv̂acq et al. [4] we adopted a data parallel approach mapping individual ants to thread blocks. Roulette wheel selection was replaced by a new parallel proportionate selection algorithm we called Double-Spin Roulette (DS-Roulette) which significantly reduced the running time of tour construction. Our solution executed up to 82x faster than the sequential counterpart and up to 8.5x faster than the best existing parallel GPU implementation.

Uchida et al. [11] implement a GPU implementation of AS and also use a data parallel approach mapping each ant to a thread block. Four different tour construction kernels are detailed and a novel city compression method is presented. This method compresses the list of remaining cities to dynamically reduce the number of cities to check in future iterations of tour construction. The speedup reported for their hybrid approach is around 43x faster than the sequential implementation (see [6]). Uchida et al. conclude that further work should be put into nearest neighbour techniques (candidate sets) to further reduce the execution times (as their sequential implementation does not use candidate sets).

We can observe that the fastest speedups are obtained when using a data parallel approach; however none of the current implementations ([3],[2],[4],[11]) use candidate sets and as a result fail to maintain speedups for large instances of the TSP. In conclusion, although there has been considerable effort put into improving candidate set algorithms (e.g. [5],[10],[9],[1]), there has been little research into developing parallel GPU implementations.

4 Implementation

In this section we present three parallel AS algorithms utilising candidate sets for execution on the GPU. The first uses a simple ant-to-thread mapping to check if this approach is suitable for use with candidate sets. The second and third implementations use a data parallel approach. The following implementations will only focus on the tour construction phase of the algorithm as we have previously shown how to implement the pheromone update efficiently on the GPU [3].

City data is first loaded into memory and stored in an $n \times n$ matrix. 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. The candidate set is then generated and transferred to the GPU. For each city we save the closest 20 cities (as recommended by Dorigo and Stützle [7]) into an array. After initialization the pheromone matrix is artificially seeded with a tour generated using a greedy search as recommended in [7].

4.1 Tour Construction Using a Candidate Set

In Fig. 1 we give the pseudo-code for iteratively generating a tour using a candidate set based upon the implementation by Dorigo and Stützle [7]. First, an ant is placed on a random initial city; this city is then marked as visited in a *tabu* list. Then for $n - 2$ iterations (where n is the size of the TSP instance) we select the next city to visit. The candidate set is first queried and a probability of visiting each closely neighbouring city is calculated. If a city has previously been visited, the probability of visiting that city in future is 0. If the total probability of visiting any of the candidate set cities is greater than 0, we perform roulette wheel selection on the set and pick the next city to visit. Otherwise we pick the best city out of all the remaining cities (where we define the best as having the largest pheromone value).

```
procedure ConstructSolutionsCandidateSet
  $tour[1]$ ← place the ant on a random initial city
  $tabu[1]$ ← visited
  for $j = 2$ to $n - 1$ do
    for $l = 1$ to 20 do
      $probability[l]$ ← CalcProb($tour[1 \dots j - 1]$,$l$)
    end-for
    if probability $> 0$ do
      $tour[j]$ ← RouletteWheelSelection($probability$)
      $tabu$[tour[$j$]] ← true
    else
      $tour[j]$ ← SelectBest($tabu$)
      $tabu$[tour[$j$]] ← true
    end-if
  end-for
  $tour[n]$ ← remaining city
  $tour\_cost$ ← CalcTourCost($tour$)
end
```

Fig. 1. Overview of an ant's tour construction using a candidate set

4.2 Task Parallelism

Although it has previously been shown that using a data parallel approach yields the best results ([3],[2],[4],[11]), it has not yet been established that this holds when using a candidate set. Therefore our first parallelization strategy considers this simple mapping of one ant per thread (*task parallelism*). Each thread (ant) in the colony executes the tour construction method shown in Fig. 1. There is little sophistication in this simple mapping, however we include it for completeness. Cecilia et al. [2] note that implementing ACO using task parallelism is not suited to the GPU. From our experiments we can observe that these observations still persist when using a candidate set and as a result yield inadequate results which were significantly worse than those obtained by the CPU implementation. We can therefore conclude that the observations made by Cecilia et al. [2] hold when using candidate sets.

4.3 Data Parallelism

Our second approach uses a data parallel mapping (one ant per thread block). Based on the previous observations made when implementing a parallel roulette wheel selection algorithm [3] we found that using warp level primitives to avoid branching lead to the largest speedups. In DS-Roulette each warp independently calculates the probabilities of visiting a set of cities. These probabilities are then saved to shared memory and one warp performs roulette wheel selection to select the best set of cities. Roulette wheel selection is then performed again on the subset of cities to select which city to visit next [3]. This process is fast as we no longer perform reduction across the whole block and avoid waiting for other warps to finish executing. As we no longer need to perform roulette wheel selection across all cities, DS-Roulette is unsuitable for use with a candidate set. However, if we reverse the execution path of DS-Roulette we can adapt the algorithm to fit tour selection using a candidate set (see Fig. 2). Instead of funnelling down all potential cities to perform roulette wheel on one warp of potential cities, we first perform roulette wheel selection across the candidate set and scale up to all available cities if no neighbouring cities are available. Our new data parallel tour selection algorithm consists of three main stages.

The first stage uses one warp to calculate the probability of visiting each city in the candidate set. An optimisation we apply when checking the candidate set is to perform a warp ballot. Each thread in the warp checks the city against the tabu list and submits this value to the CUDA operation __ballot(). The result of the ballot is a 32-bit integer delivered to each thread where bit n corresponds to the input for thread n. If the integer is greater than zero then unvisited cities remain in the candidate set and we proceed to perform roulette wheel selection on the candidate set. Using the same warp-reduce method we previously used in [3] we are able to quickly normalize the probability values across the candidate set warp, generate a random number and select the next city to visit without communication between threads in the warp. We found experimentally that using a candidate set with less than 32 cities (1 warp) was actually detrimental to the performance of the algorithm. Scaling the candidate set up from 20 cities to 32 cities allows all threads within the warp to follow the same execution path.

In stage two the aim is to narrow down the number of remaining available cities. We limit the number of threads per block to 128 and perform tiling across the block to match the number of cities. Each warp then performs a modified version of warp-reduce [3] to find the city with the best pheromone value using warp-max. As each warp tiles it saves the current best city and pheromone value to shared memory. Using this approach we can quickly find four candidates (1 best candidate for each of the warps and as there are 128 threads with 32 threads per warp) for the city with the maximum pheromone value for the final stage of the algorithm using limited shared memory and without block synchronisation.

Finally we use one thread to check which of the four previously selected cities has the largest pheromone value, visit this city and save the value to global memory. The three stages of the algorithm are illustrated in Fig. 2.

4.4 Data Parallelism with Tabu List Compression

Section 3 details the recent work of Uchida et al. [11] presenting a novel tabu list compresssion technique. A tabu list can be represented as an array of integers with size n. When city i is chosen, the algorithm replaces city tabu[i] with city tabu[$n-1$] and decrements the list size n by 1. Cities that have previously been visited will not be considered in future iterations thus reducing the search space. By adding tabu list compression to our data parallel tour construction kernel we aim to further reduce the execution time. However, as a complete tabu list is stil required for checking against the candidate set we must use two tabu lists. The second list maintains the positions of each city within the first candidate list.

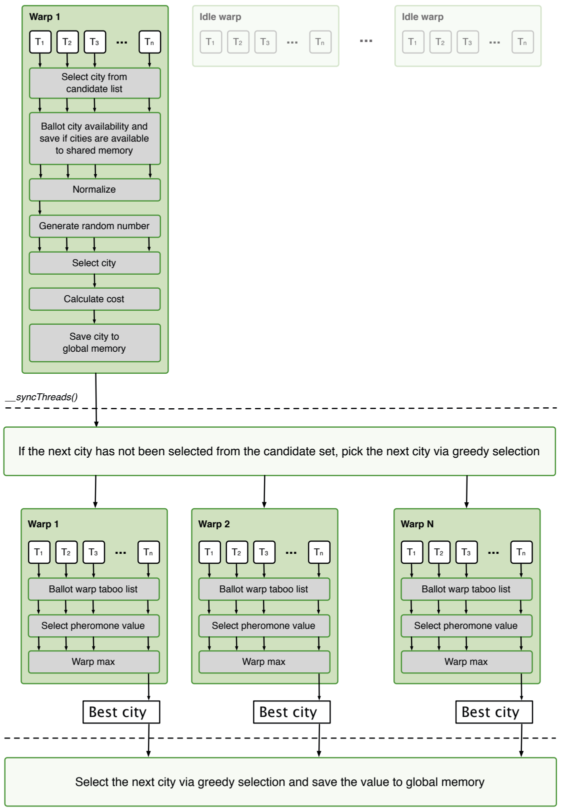
Warp 1
T1 T2 T3 ... Tn
Select city from candidate list
Ballot city availability and save if cities are available to shared memory
Normalize
Generate random number
Select city
Calculate cost
Save city to global memory
Idle warp
Idle warp
__syncThreads()
If the next city has not been selected from the candidate set, pick the next city via greedy selection
Warp 1
Warp 2
Warp N
Ballot warp taboo list
Select pheromone value
Warp max
Best city
Select the next city via greedy selection and save the value to global memory

Fig. 2. An overview of the data parallel candidate set tour construction algorithm

5 Results

In this section we present the results of executing various instances of the TSP on our two data parallel candidate set implementations and compare the results to the sequential counterpart and our previous GPU implementation. We use the standard ACO parameters but increase the candidate set size from 20 to 32 (see Section 4). The solution quality obtained by our implementations was able match and often beat the quality obtained by the sequential implementation. Our test setup consists of a GTX 580 GPU and an i7 950 CPU. Timing results are averaged over 100 iterations of the algorithm with 10 independant runs.

In Table. 1 we present the execution times (for a single iteration) of the tour construction algorithm using a candidate set for various instances of the TSP. Columns 5 and 6 show the speedup of the two data parallel implementations over the CPU implementation using a candidate set. The CPU results are based on the standard sequential implementation ACOTSP (source available at [6]) and the two GPU columns correspond to the two proposed data parallel candidate set implementations in Section 4.

Table 1. Average execution times (ms) when using AS and a candidate set

Instance	CPU	GPU 1	GPU 2	Speedup GPU 1	Speedup GPU 2
*d*198	6.39	0.77	0.85	8.31x	7.53x
*a*280	13.44	1.59	2.04	8.42x	6.59x
*lin*318	18.60	1.90	2.07	9.74x	8.99x
*pcb*442	42.37	3.67	3.96	11.55x	10.69x
*rat*783	168.90	12.13	14.49	13.92x	11.66x
*pr*1002	278.85	19.76	26.34	14.10x	10.58x
*nrw*1379	745.59	42.37	68.78	17.60x	10.84x
*pr*2392	2468.40	131.85	393.98	18.72x	6.27x

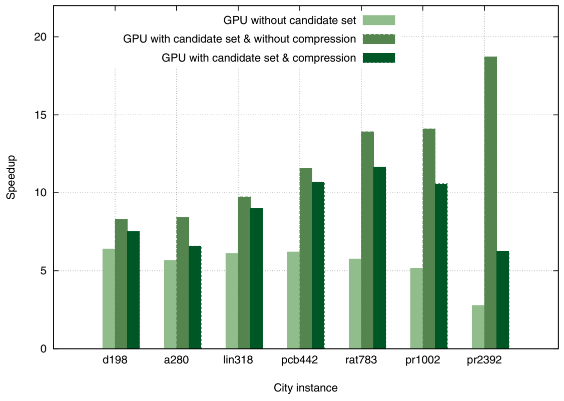

Fig. 3. Execution speedup using multiple GPU and CPU instances

Our results show the first data parallel GPU implementation achieves the best speedups across all instances of the TSP. Both data parallel approaches consistently beat the results obtained for the sequential implementation. The speedup obtained by the first data parallel implementation increased as the tour sizes increased. This is in contrast to our previous GPU implementation [3] in which the speedup reduced due to shared memory constraints and failed to maintain speedups against the sequential implementation when using a candidate set.

The results attained for the second data parallel implementation using tabu list compression show the implementation was not able to beat the simpler method without compression. As mentioned in Section 4 to implement tabu list compression, a second tabu list must be used to keep the index of each city in the first list. We observed the process of updating the second list for each iteration (for both the greedy search stage and proportionate selection on the candidate set stage) outweighed the benefits of not checking the tabu values for previously visited cities. We also observed that the increased shared memory requirements for larger instances reduced the performance of the solution.

In Fig. 3 we compare the speedup of our previous GPU implementation [3] without a candidate set against our data parallel GPU solutions. We can observe that for large instances, the speedup obtained from our GPU implementation with a candidate set increases as opposed to instances without a candidate set.

6 Conclusions

In this paper we have presented three candidate set parallelization strategies and shown that candidate sets can be used efficiently in parallel on the GPU. Our results show that a data parallel approach must be used over a task parallel approach to maximize performance. Tabu list compression was shown to be ineffective when implemented as part of the tour construction method and was beaten by the simpler method without compression. Our future work will aim to implement alternative candidate set strategies including dynamically changing the candidate list contents and size.

References

1. Blazinskas, A., Misevicius, A.: Generating high quality candidate sets by tour merging for the traveling salesman problem. In: Skersys, T., Butleris, R., Butkiene, R. (eds.) ICIST 2012. CCIS, vol. 319, pp. 62–73. Springer, Heidelberg (2012)
2. Cecilia, J.M., García, J.M., Nisbet, A., Amos, M., Ujaldon, M.: Enhancing data parallelism for ant colony optimization on GPUs. J. Parallel Distrib. Comput. 73(1), 42–51 (2013)
3. Dawson, L., Stewart, I.: Improving Ant Colony Optimization performance on the GPU using CUDA. In: 2013 IEEE Congress on Evolutionary Computation (CEC), pp. 1901–1908 (2013)
4. Delèvacq, A., Delisle, P., Gravel, M., Krajecki, M.: Parallel ant colony optimization on graphics processing units. J. Parallel Distrib. Comput. 73(1), 52–61 (2013)
5. Deng, M., Zhang, J., Liang, Y., Lin, G., Liu, W.: A novel simple candidate set method for symmetric tsp and its application in max-min ant system. In: Advances in Swarm Intelligence, pp. 173–181. Springer (2012)
6. Dorigo, M.: Ant Colony Optimization - Public Software, `http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html` (last accessed July 31, 2013)
7. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
8. NVIDIA: CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (last accessed July 31, 2013)
9. Rais, H.M., Othman, Z.A., Hamdan, A.R.: Reducing iteration using candidate list. In: International Symposium on Information Technology, ITSim 2008, vol. 3, pp. 1–8. IEEE (2008)
10. Randall, M., Montgomery, J.: Candidate set strategies for ant colony optimisation. In: Proceedings of the Third International Workshop on Ant Algorithms, ANTS 2002, pp. 243–249. Springer, London (2002)
11. Uchida, A., Ito, Y., Nakano, K.: An efficient gpu implementation of ant colony optimization for the traveling salesman problem. In: 2012 Third International Conference on Networking and Computing (ICNC), pp. 94–102 (2012)