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@durham.ac.uk, 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) [8] 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 [8] 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 [8] 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. 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 (a collection of processors constitutes a colony of ants). The coarse grained approach maps an entire colony of ants to a processing element (often augmented with a method of communicating between the colonies) [8]. Multiple colonies are executed in parallel, potentially reducing the number of iterations before termination. Dorigo and Stützle note [8] that in the case where there is no communication between colonies this is comparable to undertaking multiple independent runs of the algorithm. Adopting a coarse grained approach with no communication is considered an easy way to implement ACO in parallel without dealing with the communication overhead between colonies.

NVIDIA CUDA is a parallel programming architecture for developing general purpose applications for direct execution on the GPU [12]. CUDA exposes the GPU's massively parallel architecture so that parallel code can be written to execute much faster than its optimized sequential counterpart. CUDA compatible GPUs can now be found in wide range of devices from desktop computers to laptops and more recently mobile devices such as tablet computers and phones [13]. 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 this section we define the TSP, how the TSP can be solved using the AS algorithm and how the execution time can be reduced significantly through the use of a candidate set. 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 [7, 8, 11, 17, 18].

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. Throughout this paper we only ever consider *symmetric* instances of the TSP where $d_{i,j} = d_{j,i}$, for every edge (i, j).

The AS algorithm arose after three initially-proposed ant algorithms [8] 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.

```
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 [8] 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 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 [8], 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. 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 according to the random proportional rule and becoming an edge in some subsequent tour.

2.1 Candidate sets

Randall and Montgomery [16] note that 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 for each iteration of tour construction using a subset of cities which we refer to as a candidate set. In the case of the TSP, a candidate set contains a set of the nearest neighbouring cities for each city. This exploits an observation that optimal solutions can often be found by only visiting close neighbouring cities for each city [8]. For the TSP the candidate set can be generated prior to execution using the distances between cities and does not change during execution. In the tour construction phase (for any given city) 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 [8] utilise greedy selection to pick a city with the highest pheromone value. Randall and Montgomery [16] 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

In 2007 NVIDIA introduced CUDA, a parallel architecture designed for executing applications on both the CPU and GPU, along with a new generation of GPUs G80) with dedicated silicon to facilitate future parallel programming [10]. 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.

2.3 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()*.

2.4 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 [14]. 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 [10], 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*.

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 Delv̀acq et al. [4] we adopted a data parallel approach mapping individual ants to thread blocks and moved both the tour construction and pheromone update stages to the GPU. Roulette wheel selection was replaced by a new highly 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. However this relative speedup is greatly reduced when comparing the running time of our parallel solution against a sequential implementation utilising a candidate set. The effect of using a candidate set is most noticeable for large instances of the TSP. For the instance pr2392 [9] our GPU implementation is only around 3x faster than the sequential implementation using a candidate set.

Uchida et al. [19] 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 (which uses components from the three other kernels) is around 43x faster than the standard 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],[19]) 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],[16],[15],[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 extend our previous work [3] and 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 in parallel on the GPU [3].

4.1 Setup

Before beginning the search the 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 (an established optimization, detailed in [8], as these values do not change during each iteration of the algorithm). 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 [8]) into an array. This allows each ant to quickly check the closest cities without having to regenerate the list and sort the neighbouring cities in each iteration. Once the initialization is complete, the pheromone matrix is artificially seeded with a tour generated using a greedy search as recommended in [8].

4.2 Tour construction using a candidate set

In Section 2 we briefly described how a candidate set can be used for tour construction and in Fig. 2 we give the pseudo-code for iteratively generating a tour. This method is based upon the implementation by Dorigo and Stützle [8].

First, an ant is placed on a random initial city; this city is then marked as visited in a *tabu* list. By using a tabu list we are able to check if a given city has yet to be visted in the current tour in $O(1)$ time. 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
  $tour[1] \leftarrow$ place the ant on a random initial city
  $tabu[1] \leftarrow$ visited
  for $j = 2$ to $n-1$ do
    for $l = 1$ to $20$ do
      $probability[l] \leftarrow$ CalcProb($tour[1 \ldots j-1]$,$l$)
    end-for
    if probability $> 0$ do
      $tour[j] \leftarrow$ RouletteWheelSelection($probability$)
      $tabu$[tour[$j$]] $\leftarrow$ true
    else
      $tour[j] \leftarrow$ SelectBest($tabu$)
      $tabu$[tour[$j$]] $\leftarrow$ true
    end-if
  end-for
  $tour[n] \leftarrow$ remaining city
  $tour\_cost \leftarrow$ CalcTourCost($tour$)
end
```

Fig. 2. Overview of an ant's tour construction using a candidate set

4.3 Task parallelism

Although it has previously been shown that using a data parallel approach yields the best results ([3],[2],[4],[19]), it has not yet been established that this also holds when using a candidate set. Therefore our first parallelization strategy considers this simple mapping of one ant per thread (*task parallelism*). This allows us to directly implement the sequential version without worrying about thread cooperation. Each thread (ant) in the colony executes the tour construction method shown in Fig. 2. 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.

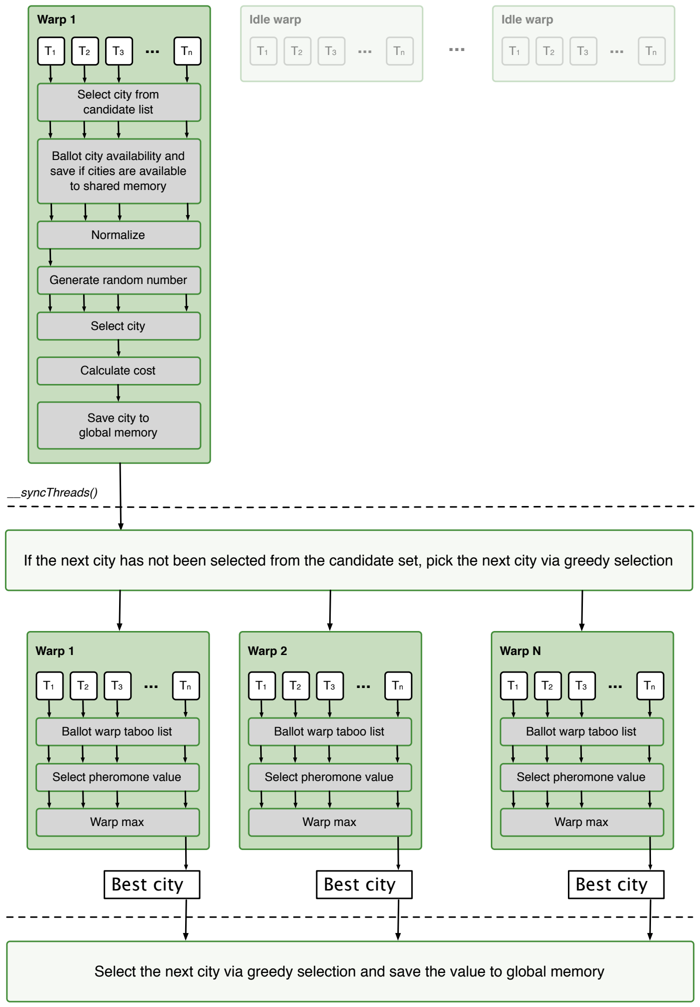
4.4 Data parallelism

Having established that task based parallelism is unsuitable when using candidate sets, 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 and avoiding branching (where possible) 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 (see [3] for additional details). 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. 3). 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 to pick which city to visit next. 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 and avoid divergence. If during this first stage of the algorithm we have selected a city from the candidate set then we skip the following two stages and repeat the process for all remaining cities in the tour.

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. Warp-reduce performs reduction across a warp with implicit synchronisation and as a result no branching. By modifiying warp-reduce, we can find the maximum value across the warp quickly with no branching. 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 very limited shared memory and without inter-warp synchronisation.

Finally we use one thread to simply check which of the four previously selected cities has the largest pheromone value and visit this city. The value is saved to global memory and the distance between the next city and the previous city is also recorded. The three stages of the algorithm are illustrated in Fig. 3.



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

4.5 Data parallelism with tabu list compression

In Section 3 we detailed the recent work of Uchida et al. [19]. In their paper they present a novel tabu list compresssion method which dynamically reduces the number of cities to check in future iterations of the algorithm. A tabu list can be represented as a simple array of integers with size n. When city i is chosen, the compression algorithm replaces city tabu$[i]$ with city tabu$[n-1]$ and decrements the list size n by 1. During runtime, the tabu list length will decrease by 1 for each iteration of the tour construction kernel thus dynamically reducing the tabu list length. By compressing the list, cities that have previously been visited will not be considered in future iterations. This reduces the search space and the total execution time. By adding tabu list compression to our data parallel tour construction kernel we aim to further reduce the execution time by reducing the work for the kernel each iteration. However, as a complete tabu list is stil required for checking against the candidate set we must allocate two tabu lists to maintain the $O(1)$ lookups essential to the speed of the candidate set solution. The second list maintains the positions of each city within the first candidate list. This secondary list must be updated each time the first tabu list is compressed. If we were to rely upon a single tabu list with compression to check if a city in the candidate set has been visited then each thread would have to iterate through the entire list.

5 Results

In this section we present the results of executing various instances of the TSP on our two data parallel GPU candidate set implementations and compare the results to the sequential counterpart and our previous GPU implementation. As we mentioned in Section 4 our task parallel implementation was unable to match the performance of the CPU implementation and offers no practical benefits. We use the standard ACO parameters but increase the candidate set size from 20 to 32 (see Section 4). In this paper we have focussed on strategies to optimise tour construction on the GPU using a candidate set and the AS algorithm. The solution quality obtained by our data parallel implementations was able match and often beat the quality obtained by the sequential implementation. As we previously noted [3] the quality of tours obtained by AS are not optimal and can be improved with local search.

Our test setup consists of an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield). The GPU contains 580 CUDA cores and has a 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 in Ubuntu 13.04. Timing results are averaged over 100 iterations of the algorithm with 10 independant runs.

5.1 Benchmarks

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. We use the same instances of the TSP as [2] and in our previous work [3]. 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
$d198$	6.39	0.77	0.85	8.31x	7.53x
$a280$	13.44	1.59	2.04	8.42x	6.59x
$lin318$	18.60	1.90	2.07	9.74x	8.99x
$pcb442$	42.37	3.67	3.96	11.55x	10.69x
$rat783$	168.90	12.13	14.49	13.92x	11.66x
$pr1002$	278.85	19.76	26.34	14.10x	10.58x
$nrw1379$	745.59	42.37	68.78	17.60x	10.84x
$pr2392$	2468.40	131.85	393.98	18.72x	6.27x

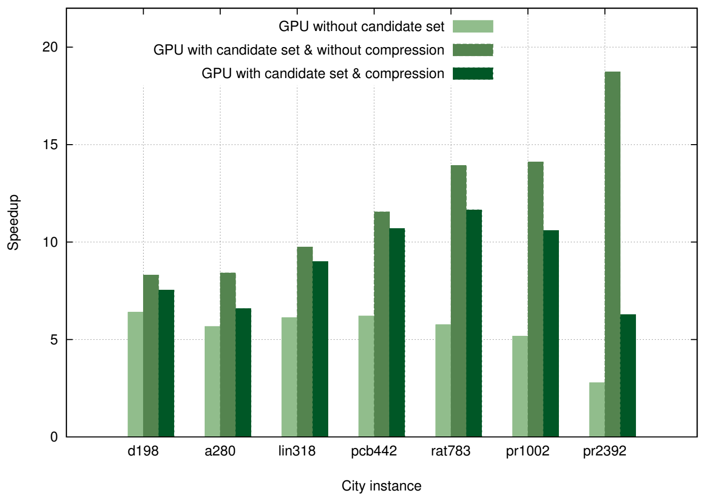


Fig. 4. Speedup of the execution of multiple GPU instances (with and without use of a candidate set) against the standard CPU implementation using a candidate set

Our results show the first data parallel GPU implementation achieves the best speedups across all instances of the TSP with speedups of up to 18x against the sequential counterpart. Both data parallel approaches consistently beat the results obtained for the sequential implementation for all instances. 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 once the city size passed the instance pr1002 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. In Table. 2 we show that tabu list compression is effective at reducing the time taken to generate a simple greedy tour. However, 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 can also observe that the increased shared memory requirements for larger instances reduced the performance and effectiveness of the solution and this can also be seen in Table. 2.

Table 2. Average execution times (ms) for a GPU implementations of the greedy search stage of our data parallel implementation both with and without tabu list compression

Instance	GPU No compression	GPU With compression
$d198$	1.38	1.09
$a280$	3.35	2.40
$lin318$	4.09	3.12
$pcb442$	9.88	6.96
$rat783$	48.65	35.61
$pr1002$	93.22	73.07
$pr2392$	1108.91	1854.31

In Fig. 4 we compare the speedup of our previous GPU implementation [3] without a candidate set against our data parallel GPU solutions. We ommit comparisons with other data parallel GPU implementations (see Section. 3) as we have previously shown our GPU implementation to be the fastest to date. We can observe that for large instances of the TSP in Fig. 4 the speedup obtained GPU implementation without a candidate set reduces whilst the opposite can bee seen for our new data parallel implementation without tabu compression.

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 the performance of the tour construction algorithm. Our best data parallel algorithm was able to achieve speedups of up to 18x the sequential counterpart. 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: Information and Software Technologies, pp. 62–73. Springer (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: Evolutionary Computation (CEC), 2013 IEEE Congress on. 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: 31/07/2013
7. Dorigo, M.: Optimization, Learning and Natural Algorithms. Ph.D. thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy (1992)
8. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
9. Heidelberg, R.K.U.: TSPLIB (Jan 2013), http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (last accessed: 31/07/2013
10. Kirk, D., Hwu, W.M.W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
11. Manfrin, M., Birattari, M., Stützle, T., Dorigo, M.: Parallel ant colony optimization for the traveling salesman problem. In: Dorigo, M., Gambardella, L.M., Birattari, M., Martinoli, A., Poli, R., Stützle, T. (eds.) Fifth Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS). Lecture Notes in Computer Science, vol. 4150, pp. 224–234. Springer Verlag (2006)
12. NVIDIA: CUDA C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (last accessed 31/07/2013)

13. NVIDIA: CUDA for ARM Platforms is Now Available, https://developer.nvidia.com/content/cuda-arm-platforms-now-available (last accessed: 05/08/2013
14. NVIDIA: Inside Kepler, http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0642-GTC2012-Inside-Kepler.pdf (last accessed 31/07/2013)
15. Rais, H.M., Othman, Z.A., Hamdan, A.R.: Reducing iteration using candidate list. In: Information Technology, 2008. ITSim 2008. International Symposium on. vol. 3, pp. 1–8. IEEE (2008)
16. Randall, M., Montgomery, J.: Candidate set strategies for ant colony optimisation. In: Proceedings of the Third International Workshop on Ant Algorithms. pp. 243–249. ANTS '02, Springer-Verlag, London, UK, UK (2002)
17. Stützle, T.: Parallelization strategies for ant colony optimization. In: Fifth Int. Conf. on Parallel Problem Solving from Nature (PPSN-V). pp. 722–731. Springer-Verlag (1998)
18. Stützle, T., Hoos, H.H.: MAX-MIN ant system. Future Gener. Comput. Syst. 16(9), 889–914 (2000)
19. Uchida, A., Ito, Y., Nakano, K.: An efficient gpu implementation of ant colony optimization for the traveling salesman problem. In: Networking and Computing (ICNC), 2012 Third International Conference on. pp. 94–102 (2012)