Ant Colony Optimization Parallel Algorithm for GPU

Honours Project - COMP 4905

Carleton University

Karim Tantawy

100710608

Supervisor: Dr. Tony White, School of Computer Science

April 10th 2011

Abstract

In nature, ants in a colony work together to forage for food by laying pheromone trails as guides towards food sources. The Ant Colony Optimization algorithm works by the same principle: simulated ants construct solutions to decision problems, and pheromone values are updated to favour better solutions. We apply this to the Travelling Salesman Problem, where the goal is to find the shortest tour between a set of cities such that each city is visited exactly once. Ant Colony Optimization is well suited to this problem due to the large search space, for example with 50 cities, the number of possible tours exceeds the number of atoms in the known universe. We implement the Ant Colony Optimization algorithm using NVIDIA CUDA to run on the Graphical Processing Unit to take advantage of the parallel nature of the heuristic. GPUs allow multiple threads to run in parallel, and these threads are arranged into thread blocks. We create one thread block per ant which is responsible for maintaining the necessary state information and generating the tour. The number of threads per block becomes the critical factor, we found that 64 threads works best as CUDA issues thread instructions in groups of 32. We achieved a greater than ten factor speedup for problem instances with up to 1,291 cities for the tour construction phase.

Acknowledgements

I would like to thank Dr. Tony White for his time, ideas, advice and guidance while helping me prepare this report. I would also like to thank Dave McKenney for his ideas on achieving greater speedups.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Overview of Ant Colony Optimization

The Ant Colony Optimization (ACO) algorithm is used to model an ant colony that can solve computational problems that are reducible to finding optimal paths in a graph; similar to how some ant species forage for food. These ants deposit pheromone on the ground in order to mark more favourable paths that should be followed by other members of the colony. ACO uses this mechanism to optimize decision problems. This algorithm was first proposed by Marco Dorigo in his PhD thesis[1].

Deneubourg et al.[3] studied the biological foundations that govern the pheromone laying and following behaviour of ants. In the now famous “double bridge experiment”, a colony of Argentine ants was connected to a food source by equal length bridges. With this setup, the ants would start to explore the area around the nest until they located the food source by means of either bridge. Once they located the food source, they deposited a pheromone trail along the path back to the nest. When initially encountering the two bridges, each ant would indiscriminately choose a path, however due to random fluctuations, after some time one of the two bridges would build up a higher concentration of pheromone. This build up in turn attracts more ants, and the positive feedback loop results in the eventual convergence of the entire colony towards using the same bridge.

This positive feedback mechanism can be exploited by the ant colony as a whole to find the shortest path to a food source. Goss et al. [4] constructed a variant of the double bridge experiment where one bridge is significantly shorter than the other. In this setup, ants that chose the shorter bridge are first to reach the food source and return back to the nest while depositing pheromones ahead of the ants that chose the longer bridge. This increases the chance that subsequent ants choose the shorter bridge over the longer one. Pheromones also

evaporate over time, thus, shorter paths that take less time to travel end up receiving more pheromones via re-enforcement compared to longer paths that take more time to traverse. In this setup, the entire colony quickly converges to using the shorter bridge to reach the food source.

Applications of ACO to Travelling Salesman Problem

The Travelling Salesman Problem (TSP) involves a set of cities with known distances between them. The goal is to construct the shortest tour that visits each city exactly once. ACO can be used to find an optimal solution to TSP. The set of cities can be considered a graph where each city is represented by a vertex, and the weight of each edge is the distance between the two cities. Each edge also contains information regarding its pheromone level. The method is to simulate a number of artificial ants, which perform a series of distributed searches on the graph. The ants are randomly placed on a starting city, complete a tour by visiting all other cities while regarding two main criteria. The first, is that each city must be visited exactly once. The second is that the pheromone level of an edge effects the probability of traversal. After the ants have completed their tours, the pheromone levels on each edge are updated by considering each ant separately. The edges it traversed have their pheromone levels increased by an amount, if any, that is determined by the quality of the solution relative to those obtained by other ants. The ants follow a heuristic edge selection mechanism based on the distance between the two cities and the pheromone level. Such a heuristic tends towards an optimal solution, however a process of simulated pheromone evaporation must be applied to stop the process from stalling in a local minimum. This is achieved by reducing the pheromone on all edges by a small amount after each iteration of tours is completed. As better tours lay more pheromone, this biases ants in future iterations to construct solutions similar to the previous best ones.

ACO algorithms have proven to be effective at finding solutions for instances of TSP. Although the best solution is not always necessary (as most good solutions suffice for practical

applications), convergence to the optimal solution has been shown. We consider optimizing the running time of the ACO algorithm by using Graphics Processing Units to take advantage of the parallel nature of the problem and the unique hardware architecture.

Overview of Graphics Processing Units

The physical design of the Graphics Processing Unit (GPU) is much different than the Computational Processing Unit (CPU). The GPU is designed such that more transistors are devoted to data processing rather than data caching and control flow, as illustrated in Figure 1[5]. Thus the GPU is specialized for parallel high-intensity computations.

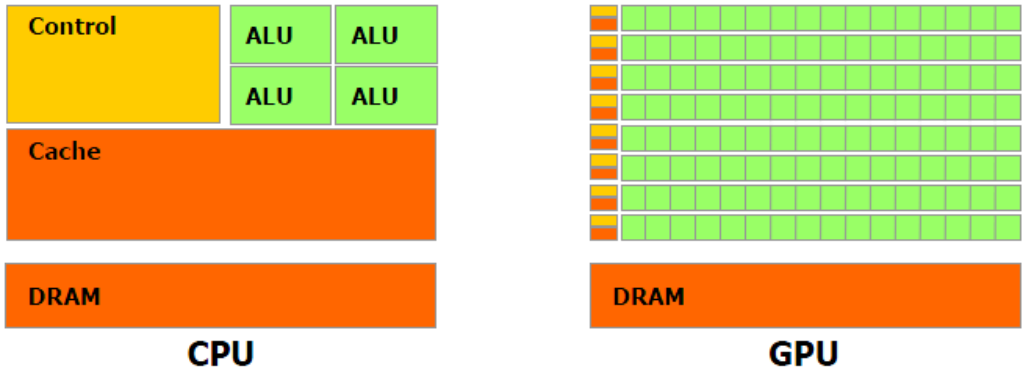

FIGURE 1. CPU vs. GPU architecture

Problems that can be expressed as data-parallel computations are much more suited for running on the GPU compared to the CPU. The tour-construction phase of the ACO algorithm lends itself nicely as a highly parallel sub-problem suitable for GPU implementation. Each ant can be thought of as an autonomous agent, and the whole colony of ants can construct tours in parallel without relying on information from each other. The only information exchange that takes place, the pheromone update, is done after all ants have constructed a solution.

NVIDIA CUDA

The Compute Unified Device Architecture (CUDA) was released in 2006 by NVIDIA, it is the computing engine found on their GPUs. CUDA allows software developers to interface with GPU devices and write code that runs directly on them through extensions to various programming languages such as C, Python, Perl, Fortran, and Java[6].

Kernels

CUDA allows the programmer to write functions called *kernels* that are run on the GPU. However, unlike normal functions that only run once when called, the kernel is executed *N* times in parallel by *N* different CUDA threads, where the value of *N* is specified by the programmer. Each thread that executes in a kernel is given a unique *thread ID* value that is accessible within the kernel[6]. For example, consider a simple program that adds two vectors, *A* and *B* of size *N*, and saves the result into vector *C* (also of size *N*). This can be defined as a kernel with *N* threads; each thread is responsible for the position in the vector corresponding to its unique *thread ID*. Thus the first thread would read the values of the first positions in vectors *A* and *B*, add them, and save the result into the first position in vector *C*. On a CPU or GPU, this will always take *N* additions. However, the CPU must perform them sequentially while the GPU threads perform all additions in parallel.

Thread Blocks

Threads can be grouped together to form *thread blocks*. The threads can be mapped as one, two, or three-dimensional blocks. For example, to add two matrices of size *N* x *N*, a kernel similar to the vector addition can be invoked using an *N* x *N* thread block. There is a limit to the number of threads per block, since all threads needs to reside on the same processor core and must share the limited resources of that core. Currently, devices with compute capability 2.0 and above support up to 1024 threads per block[6]. It is also possible to run multiple instances

of a kernel with equally shaped thread blocks, allowing the total number of threads running at once to be the max number of threads per block times the number of blocks.

Memory Hierarchy

GPU devices contain multiple types of memory with specific access configurations. Each thread has access to private local memory as well as shared memory that is visible to all threads in the same thread block. Global memory is accessible to all threads for read or write transaction. It is the largest memory storage (a few gigabytes), however has the slowest access times. Constant memory is similar to global, however it is read only and much smaller in size. Any data that is used by the GPU must first be copied over from the host (i.e. CPU) to either global or constant memory. Any data that the host wishes to retrieve from the device must reside on global memory. Shared memory is the fastest type, followed by constant memory. Local memory varies, it is optimally stored in registers, however if a lot of local memory is used by a thread that it exceeds the maximum physical storage of the chip, it is transferred to global memory. Thus there are many necessary considerations for determining the optimal memory layout, as different segmentations can adversely affect the running time of a program.

ACO META-HEURISTIC

The general idea of the ACO meta-heuristic is to let artificial ants construct solutions by a sequence of probabilistic decisions. This is done in an iterative process where the good solutions found by the ants of an iteration should guide the ants of following iterations by updating their associated pheromone values. The process continues until a stopping criteria is satisfied, such as the number of iterations or the quality of a solution. The algorithm can be described roughly as follows:

Algorithm 1: ACO Meta-heuristic

Initialize parameters, variables, and pheromone values

while termination condition is not met **do**

for each Ant

Construct a solution

end for

Update pheromone values

end while

When applying ACO to TSP we consider a set of *n* cities with distances d_{ij} between each pair of cities *i, j*. The pheromone values are encoded as a *n* x *n* matrix where T_{ij} is the pheromone value of the path between the cities *i*, *j*. Since each city can only be selected once, we denote *S* to be the set of selectable cities remaining, thus we can describe the probability of selecting city *j* after city *i* as:

$$Pij = \frac{Tij}{\sum_{z \in S} Tiz} \; \forall \, j \in S$$

This only considers the pheromones values, however there is more information that can help guide the ants in constructing a tour. This heuristic information (such as the distances between the cities) is denoted by N_{ij}. Combining both and assigning the values of α and β to correspond to the relative influence of pheromone and heuristic values, we can describe the probability of selecting city *j* after city *i* as:

$$Pij = \frac{\mathrm{T}_{\mathrm{ij}}^{\alpha} \cdot \mathrm{N}_{\mathrm{ij}}^{\beta}}{\sum_{z \in S} \mathrm{T}_{\mathrm{iz}}^{\alpha} \cdot \mathrm{N}_{\mathrm{ij}}^{\beta}} \; \forall \, j \in S$$

$$\alpha + \beta = 1, \text{and } \alpha, \beta \geq 0$$

Assuming there are *m* ants, after one iteration there will be *m* solutions constructed. The next step is to update the pheromone values in two phases. The first phase is evaporation, where all values are reduced by an amount *p*, the evaporation rate.

$$T_{ij} = (1 - p) \cdot T_{ij}$$

The second phase is to update the pheromone values of the edges based on the constructed solutions. Each edge is increased as follows:

$$T_{ij} \mathrel{+}= \sum_{k=1}^{m} \Delta T_{ij}^{k}$$

$$\Delta T_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ used edge } (i,j) \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

The length of the tour constructed by ant *k* is L_k, and *Q* is a constant value. In this system, all ants are given equal consideration and weighting. There have been alterations and modifications to this basic Ant System (AS), discussed next.

ANT SYSTEM VARIANTS AND LITERATURE REVIEW

One of the first improvements over the original AS introduced by Dorigo et al. is the concept of using "elitist" ants[13]. Each elitist ants remembers one of the best tours found by any ant, and every iteration increases the pheromone on the corresponding edges. Thus, the best tours are constantly reinforced, effectively narrowing the search space for other ants, and leading to faster convergence towards optimal solutions. The main drawback of the Elitist Ant

System is the sensitivity to the number of elitist ants. With too many, there is a high risk of premature convergence to a local optimum; with too few there is limited effectiveness.

Another improvement over AS is the *MAX-MIN Ant System* proposed by Stützle and Hoos, it produces better solutions in less time[7]. Only the best ant in the current cycle or the whole execution updates the pheromone values. To avoid premature search stagnation, the pheromone values have an upper limit of T_{max} and a lower limit of T_{min}. The initial pheromone values on all edges is set to T_{max}, and is decreased by evaporation after every iteration [12].

An enhancement to elitism proposed by White et al. is the use of Local Best Tour (LBT)[14]. Each LBT ant remembers the best tour it has found so far, and reinforces the associated edges after each iteration. LBT helps overcomes the premature convergence of elitism, and generally outperforms with regards to solution quality. LBT is also suitable for distributed and parallel executions, because no global knowledge transfer is needed.

Guntsch and Middendorf proposed another approach using population based ACO[9]. Population based ACO maintains a population *P* of the *k* best tours that have been found in past iterations, and uses them to update the pheromone matrix. After every iteration, the best tour found is added to the best tour population. Once the population reaches the limit *k*, the oldest tour is removed when adding a new one.

An early parallelization techniques implemented by Bolondi and Bondaza uses multiple processors where each one is responsible for a single ant[11]. Due to the high communication overhead between the processors, it did not scale very well. Later attempts by Talbi et al.[15] at improving the results involved the use of master-worker approach. Each worker processor is

responsible for a single ant, and all the workers send their solutions to the master process which updates the pheromone matrix and updates the best solution found.

GPU Parallelization Strategy

As mentioned earlier, since each ant constructs a tour independently of other ants, the tour construction phase is a good sub-problem for parallelization. The most straightforward implementation is to create a GPU thread for each ant. This thread is responsible for maintaining all the necessary information the ant needs, mainly a *tabu list* of potential cities to visit. The thread checks which cities can be visited, calculates the heuristic and pheromone values for those cities, and probabilistically selects the next one to visit. The number of cities in each TSP instance can vary greatly, as well as the number of ants simulated. It is recommended to use the same number of ants as cities[8], so an instance with 100 cities would need 100 simulated ants to be solved effectively. Creating one thread per ant is a rather naive parallelization method, and it severely under utilizes the GPU device.

The approach we consider is to assign a thread block for each ant. The thread block has one thread per city in the TSP instance. Each thread is assigned the task of checking the *tabu list* to determine if its corresponding city can be visited, and if so, calculates the associated heuristic and pheromone values. Thus with n cities, the workload is split by a factor of $1/n$. The next city is chosen probabilistically as before, and the *tabu list* is updated accordingly. With one thread per city, there is a design limitation that restricts the maximum instance size to 1024, as thread blocks have a maximum size of 1024.

The next improvement is to avoid re-calculating the heuristic and pheromone values at each step in the tour construction phase. At the beginning of each new iteration, we create a

total matrix, and populate it with the result of [Equation 2]. Thus $Total_{ij}$ is the probability of selecting city *j*, after city *i*. Each thread considers *i* to be the current city of the tour, and *j* to be the city it corresponds to. If its corresponding city has already been visited, the probability is obviously set to 0. Thus all the heuristic and pheromone values are only calculated once per kernel call.

With one thread per city large instance sizes do not scale well since many of the threads are idle for a large duration of the execution. We consider setting the number of threads per ant to 32, 64, or 96. CUDA schedules thread instructions in *warps* of size 32, thus any multiple of 32 threads is the most efficient for execution. A tiling technique is used to divide the total cities among the available threads, so that each thread is responsible for a group of cities. This allows for instances larger than the previously mentioned 1024 limit when using one thread per city.

Algorithm 2: Tour Construction

Copy total matrix to device

Generate and copy random matrix to device

Run GPU kernel

Copy tours to host

Set ant tours

This is the general outline of each iteration in the process. The total matrix changes from each iteration due to the updated pheromone values. The random matrix is re-generated and copied to provide random numbers for the stochastic selection process. Once the GPU kernel is run, the tours are copied from device global memory to the CPU, and the ant tours are set accordingly.

The ant tours are placed in global memory since they will need to be copied back to the CPU. The total and random matrices are placed in constant memory since they are read only. The tabu and probability lists are stored in shared memory for inter-block visibility between threads and low latency access times.

Algorithm 3: GPU Kernel

Initialize tabu list, probability list

for each city **do**

Check tabu list

Set probability of visiting

end for

for each ant **do**

Compute probability sum

Read random variable, choose next city to visit

Update tabu list, current city, and ant tour

end for

Parallel Reduction

The last improvement we consider is parallel reduction when computing the total probability sum. Instead of just one thread computing the sum, we parallelize the operation such that with *n* elements in an array, there are *n/2* additions in the first step with each element in the second half of the array being added to the corresponding element in the first half. This is done with *n/2* threads participating, and since there are no shared memory conflicts, all additions can be done in parallel. The process is then repeated while only considering the updated first half of the array, and eventually the first element will contain the sum of all elements. This takes $log_2 n$ steps, which is much faster than the *n* steps required

sequentially. It is also work efficient as there will be *O(n)* additions which is the minimum required.

When using a fixed number of threads, the reduction is altered slightly. Each thread keeps a running subtotal of the probability sum for its subset of cities. The subtotals are then reduced in parallel, as the reduction requires the number of threads to equal the number of values to add. For example, with 32 threads, and each thread responsible for 5 cities: each thread will calculate the subtotal for 5 cities, then all 32 subtotals are reduced using the reduction algorithm described above.

EXPERIMENTAL RESULTS

All tests where preformed with the following machine setup: Dual Core CPU @ 2.00 GHZ with NVIDIA GeForce GTX 480 GPU device with CUDA version 2.0. The full testing specifications can be found in [Appendix A]. We used the ACOTSP V 1.01 provided with the GPL license by Thomas Stützle[Appendix B] as the CPU benchmark, and modified it to incorporate our GPU code. We used the MAX-MIN Ant System as it offers improved tour results over the generic ACO implementation, as shown by Delévacq et al.[10] Since the process is stochastic, the tours created will vary from run to run as will the running times. We averaged the time of 20 runs, and found the standard deviation to be less than 5%. The length of the tours found by both CPU and GPU implementations also matched closely to each other, however the tour results were not considered in great detail.

Instance	d198	lin318	pcb442	att532	rat783	d1291	fl1557	rl1889
CPU	0.08	0.36	1.46	1.68	3.94	13.01	18.20	38.75
GPU with # of Cities Threads per Ant	0.03	0.08	0.24	0.61	3.86	-	-	-
GPU with Parallel Reduction	0.01	0.03	0.12	0.29	1.69	-	-	-
32 Threads per Ant	0.01	0.03	0.06	0.11	0.36	1.78	3.61	7.45
64 Threads per Ant	0.01	0.03	0.06	0.11	0.31	1.40	2.98	6.00
96 Threads per Ant	0.01	0.03	0.06	0.11	0.31	1.42	3.25	6.65
Total Speedup Factor for 64 Threads	8.00	12.00	12.17	15.27	12.71	15.49	6.11	6.46

Table 1. Tour Construction Times

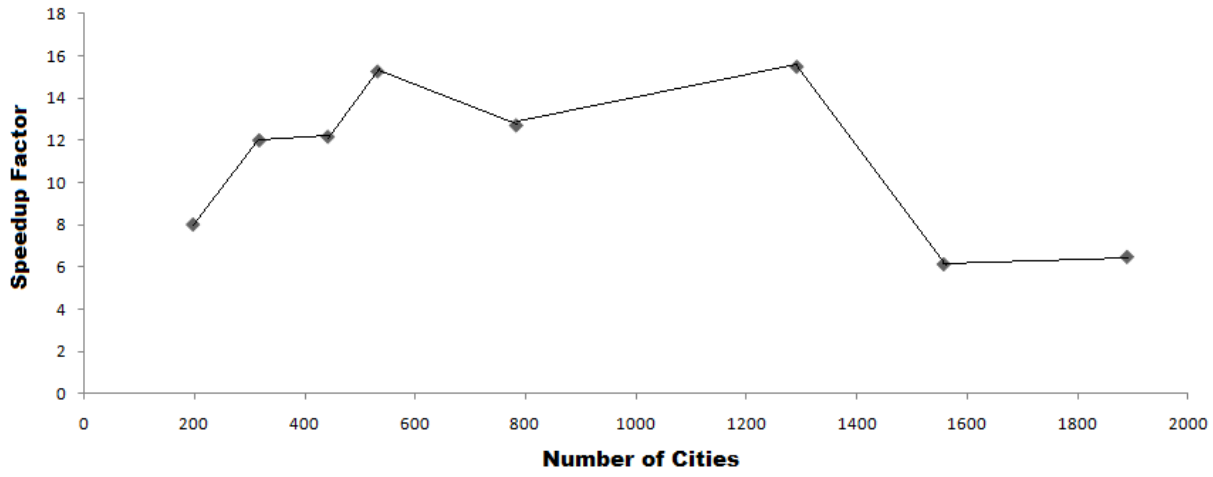

FIGURE 2. Speedup Factor vs. Number of Cities Using 64 Threads

Using 64 threads per ant gives the best speedups across the board. The speedup declines with 1557 cities, however increases slightly with 1889 cities. With up to about 600 instances using 32, 64, or 96 threads performs equally well. Using 96 threads gives slightly better results than using 32 threads for larger instances over 600 cities.

The setup phase includes generating random values, copying memory to the device, copying memory back to the host and setting the ant tours.

Instance	Setup Time	Kernel Time	Setup / Kernel
d198	<0.01	< 0.01	-
lin318	< 0.01	< 0.03	0.33
pcb442	0.01	0.05	0.20
att532	0.02	0.09	0.22
rat783	0.03	0.29	0.10
d1291	0.08	1.32	0.06
fl1557	0.12	2.86	0.04
rl1889	0.17	5.83	0.03

Table 2. Setup Time vs. Kernel Time Using 64 Threads

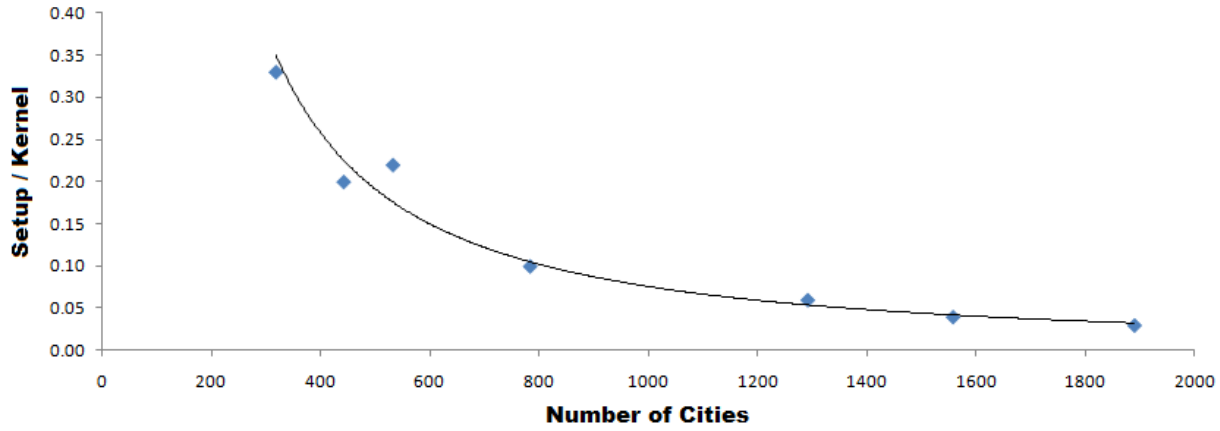

FIGURE 3. Setup / Kernel Time vs. Number of Cities Using 64 Threads

This shows that as the problem size increases, the cost of transferring the data between the CPU and GPU also increases, but becomes a negligible portion of the overall running time. With 318 cities the overhead takes about 30% of the time, and that decrease to 3% of the time with 1889 cities.

CONCLUSIONS AND FUTURE WORK

Our GPU implementation of ACO shows the best speedups when using 64 threads per ant. Using 32 or 96 threads per ant is also viable as the results are only slightly slower. Setting the number of threads per ant to equal the number of cities works well for instances of size less than 512, but does not scale well past that point. Thus the key factor in our implementation is choosing an appropriate thread to city ratio. The next step would be to investigate the speedup decline with larger problem instances. Interestingly, the speedup increase slightly between the instances with 1557 and 1889 cities. Additionally, any improvements to the kernel code would result in a speedup for all instance sizes.

ACO algorithms have many applications beyond the TSP, and are well suited for NP-hard problems. These problems generally fall into one of the following categories: routing, scheduling, or assignment. Scheduling problems are concerned with allocating resources over time, such as job shop scheduling[16]. Assignment problems require a set of objects to be assigned to a set of resources, such as planning a group of activities and assigning them to specific locations. ACO has been successfully applied to the quadratic assignment problem by Stützel and Hoos[12]. It is clear that ACO can be modified accordingly to allow its application to problems that arise across many distinct fields.

REFERENCES

[1] M. Dorigo, Optimization, learning and natural algorithms, Ph.D. Thesis, Politecnico di Milano, Italy, 1992.

[2] http://en.wikipedia.org/wiki/Ant_colony_optimization, accessed on 11/01/2011

[3] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels, "The self organizing exploratory pattern of the Argentine ant," *Journal of Insect Behavior*, vol. 3, p. 159, 1990.

[4] S. Goss, S. Aron, J.-L. Deneubourg, and J. M. Pasteels, "Self-organized shortcuts in the Argentine ant," *Naturwissenschaften*, vol. 76, pp. 579– 581, 1989.

[5] "NVIDIA CUDA C Programming Guide Version 3.2", 11/09/2010, p. 3

[6] "NVIDIA CUDA", http://en.wikipedia.org/wiki/CUDA , accessed on 01/03/2011

[7] Ant Colony Optimization, "Artificial Ants as Computational Intelligence Technique", Marco Dorigo, Mauro Birattari, Thomas Stützle. IRIDIA Technical Report, 2006.

[8] M. Dorigo and T. St¨utzle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[9] M. Guntsch and M. Middendorf. A population based approach for ACO. In *Applications of Evolutionary Computing - Proc. EvoWorkshops 2002*, Springer, LNCS 2279, 72–81 (2002).

[10] Audrey Delévacq, Pierre Delisle and Michael Krajecki, “Max-Min Ant System on Graphics Processing Units” CReSTIC, Universite de Reims Champagne-Ardenne, France, 2010

[11] M. Bolondi, and M. Bondaza: Parallelizzazione di un algoritmo per la risoluzione del problema del comesso viaggiatore; Master’s thesis, Politecnico di Milano, (1993).

[12] T. Stutzle and H. Hoos, "MAX-MIN Ant system", Future Generation Computer Systems, v. 16, n. 8, pp. 889-914, 2000.

[13] Dorigo M., Maniezzo V. and Colorni A. The Ant System: Optimization by a Colony of Cooperating Agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1):29-41, 1996.

[14] White T., Kaegi S. and Oda T. Revisiting Elitism in Ant Colony Optimization. GECCO 2003.

[15] E-G. Talbi, O. Roux, C. Fonlupt, and D. Robilliard: Parallel ant colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17(4):441-449 (2001). Preliminary version in J. Rolim et al. (Eds.) *Parallel and Distributed Processing, 11 IPPS/SPDP’99 Workshops*, LNCS 1586, Springer, 239–247 (1999).

[16] “Job shop scheduling”, http://en.wikipedia.org/wiki/Job_shop_scheduling, accessed on 01/03/2011

APPENDIX A - TESTING SETUP

Machine info:

Genuine Intel: Intel(R) Pentium(R) Dual CPU E2180 @ 2.00GHz

1200 MHz, 1024 KB cache size, 2 CPU cores, 36 bits physical, 48 bits virtual address size

GPU Device info:

Device Name : GeForce GTX 480

Total Global Memory Size : 1609760768 bytes

Total Shared Memory Per Block : 49152 bytes

Total Constant Memory : 65536 bytes

Device Major Revision Numbers : 2

Device Minor Revision Numbers : 0

Registers Per Block : 32768

Warp size : 32

Max threads per block : 1024

Multi-Processor Count : 15

APPENDIX B – ACOTSP V1.01

From the Ant Colony Optimization website by Marc Dorigo,
http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html

Author: Thomas Stützle
Short description: This software package provides an implementation of various Ant Colony Optimization (ACO) algorithms applied to the symmetric Traveling Salesman Problem (TSP). The ACO algorithms implemented are Ant System, Elitist Ant System, MAX-MIN Ant System, Rank-based version of Ant System, Best-Worst Ant System, and Ant Colony System.
Aim of the software: Provide an implementation of ACO algorithms for the symmetric TSP under one common framework. The implementation is reasonably high performing.
License: GPL
Programming language: Developed in ANSI C under Linux (no guarantees that it works nicely under Windows; however, some limited tests showed that the code also works fine under Windows and Mac OS X).
Comment: This is Version 1.01 of ACOTSP; it is in large part identical to the software used to produce the results in the book Ant Colony Optimization by Marco Dorigo and Thomas Stüzle, MIT Press, Cambridge, MA, USA, 2004. It has been slightly re-structured, adapted to make the code more readable, some more comments were added, and a new command line parser was generated with Opag, Version 0.6.4.

Our Modifications:

We modified this software package to incorporate our GPU code. The command line parser has been updated to allow for a new option to use CUDA. By adding “--cuda" or “-p" to the command line, the parallel GPU implementation will be used, otherwise the default is to use the CPU. Another change was to increase the maximum number of ants allowed from 512 to 2024.