A PARALLEL ANT COLONY OPTIMIZATION ALGORITHM FOR THE TRAVELLING SALESMAN PROBLEM: IMPROVING PERFORMANCE USING CUDA

by

Octavian Nitica

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Honors Bachelor of Science in Computer and Information Science with Distinction.

Spring 2011

A PARALLEL ANT COLONY OPTIMIZATION AGLORITHM FOR

TRAVELLING SALESMAN PROBLEM:

IMPROVING PERFORMANCE USING CUDA

by

Octavian Nitica

Approved: _____

John Cavazos, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Louis Rossi, Ph.D.
Committee member from the Department of Mathematical Sciences

Approved: _____

Pak-Wing Fok, Ph.D.
Committee member from the Board of Senior Thesis Readers

Approved: _____

Michael Arnold, Ph.D.
Director of the University Honors Program

ACKNOWLEDGMENTS

I would like to thank John Cavazos for helping me as an advisor and mentor here at Delaware. Your help has been invaluable to the development of me as a researcher.

I would like to thank Scott-Grauer Gray for helping me with CUDA programming and with the development of the multi-GPU portion of the work.

I would like to thank the UD PetaApps Cloud Physics group (especially Lou Rossi and Lian-Ping Wang) for funding me the past summer and helping me get started on this research.

Finally, I would like to thank my friends and family for their constant support and for making college an unforgettable experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The ant colony optimization (ACO) algorithm is a metaheuristic algorithm used for combinatorial optimization problems. It is a good choice for many hard combinatorial problems because it is more efficient that brute force methods and produces better solutions than greedy algorithms. However, ACO is computationally expensive, and it can still take a long time to compute a solution on large problem sets. Fortunately, the structure of the ACO algorithm is amenable to being parallelized. By using CUDA to implement an ACO algorithm, I achieved significant improvement in performance over a highly-tuned sequential CPU implementation. I propose several different ways to improve the performance of the ACO algorithm using GPUs, including a multi-GPU approach. I ran the CUDA-parallelized ACO algorithm on the travelling salesman problem (TSP) problem and compared our performance to a highly-tuned sequential CPU implementation.

Chapter 1

INTRODUCTION

The main topic of this thesis is to improve the performance of the ACO algorithm using Graphics Processing Units (GPUs). I use the CUDA platform to program Nvidia GPUs to achieve this goal, and my work involves applying ACO to the Travelling Salesman Problem (TSP). The main contribution of this thesis is a faster ACO implementation for the TSP and a framework for future improvement on this problem. In another aspect of this thesis, I explore many issues with porting such an algorithm to a GPU. It is also my goal to show that ACO algorithms are amenable to being run in parallel on GPU architectures.

The second chapter deals with background to the algorithm and the problem it applies to, and why such research might be relevant. Next, the third chapter deals with GPU computing and explains programming with CUDA. I use the well known ACOTSP package as a baseline to compare against, and I also port several stages of this package to run on the GPU. The fourth chapter explains my first implementation of the work, where I port the construction stage of the algorithm in the ACOTSP package. My second implementation is detailed in the fifth chapter, where I port all the stages of the ACO algorithm to the GPU. The sixth chapter covers my work in

applying a multi-GPU approach, where the algorithm runs on several GPUs at once to further improve execution time. The seventh chapter analyzes some related work and outlines an option for future work by applying the ACO algorithm to instruction scheduling. Finally, the thesis ends with a chapter containing my conclusions.

Chapter 2

BACKGROUND AND MOTIVATION

In this thesis a parallel ant colony optimization algorithm is applied to the Travelling Salesman Problem. This section highlights the reason behind why such an implementation is beneficial. It discusses the difficulty of solving NP-Hard problems such as the Travelling Salesman Problem. It also covers the reasoning behind search algorithms such as the ant colony algorithm and why such an algorithm might be applied. Finally, it brings up why such an algorithm might benefit from execution on parallel platforms such as GPUs.

2.1 NP-Hard Problems

Complexity classes can be thought of as problems that are relatively the same difficulty to solve in terms of resources. In this thesis, the term is used exclusively in regards to computation (or clock cycles used). The ACO algorithm was developed to solve problems belonging to the complexity class known as NP-Hard. Problems belonging to this class are often not possible to solve with brute force calculation due to the scaling of computation in regards to input. A notation called *Big O* is commonly used to represent algorithm complexity. A simple way to look at Big O notation is it is a representation of the approximate running time of an algorithm with respect to its input. Big O notation does not deal with absolute notions, but a Big O running time

denotes the upper bound of computation ignoring constants. In other words, Big O denotes the worst-case running time with respect to some input.

Since Big O is relative to the input, it uses the input size as part of the notation. Let n denote the size of input given to an algorithm. An algorithm of O(n) complexity has linear complexity. This means the algorithm runs approximately c computations for every input element where c is a constant. An example of this would be an algorithm to sum all the elements in an array. Every extra element x adds one more iteration the algorithm would have to run to add x to the total. In this case the algorithm has a c value of 1. Now an algorithm of complexity O(n^m), where m is a constant and $m > 1$, has polynomial time complexity. These types of algorithms perform at worst-case cn^m iterations to solve a problem for an input size of n, where c and m are constants.

A problem belonging to the NP complexity class means that a solution to the problem can be verified to be correct in polynomial time. NP-Hard problems are at least as hard as problems belonging to the NP complexity class. While solutions in this complexity class may be checked fairly rapidly, they may have much larger computational requirements for generating all the solutions of a problem. If all the solutions to a problem are not generated, it may be impossible to guarantee the best solution has been found. A solution with the best candidate solution value to a problem is often referred to as the *optimal* solution. A problem may have more than one optimal solution. Take for example when finding the shortest path between two

vertices in a graph: there may be two different routes that have the same lowest cost value. Many algorithms have been developed to compute good, yet not optimal, solutions to NP-Hard problems in a reasonable amount of time. For more information on algorithm complexity, please refer to the following book, *Introduction to Algorithms* [35] (pgs. 41-61, 966-986).

2.2 Search Algorithms

The type of algorithm used in this paper to find a good solution is called a search algorithm. A *search algorithm* tries to find an object with specified properties among a collection of objects. The set of all candidate solutions to a problem that a search algorithm tries to solve is called the *search space*.

I give a more formal definition here. Let $D = \{d_1, d_2, ..., d_n\}$ be a set of data elements. Then let $S = \{s_1, s_2, ..., s_m\}$ define the search space, where every element s_m is a subset of D related by some mathematical formula or procedure. A search algorithm constructs objects in the search space (also called *candidate solutions*) from the data and then finds the object s_{best} which has the best value. The method that solutions are better than others is problem specific. The basic idea is to enumerate a value for every candidate solution so that it is comparable to other solutions. Once solutions are comparable, it is a straightforward to choose the best one.

Another important thing to note about search algorithms is that the search space is often represented as a graph. Taking our definition of grouping you can think of each data element

as a node in the graph. The mathematical relation between the nodes represents the edges in the graph. Then a valid path through the graph is a candidate solution to the problem. The properties of the graph are dependent on the problem the search algorithm is attempting to solve. Looking at search spaces this way also helps in developing search algorithms. Also, for many problems, it may not even be known how many data elements are in a solution. Therefore trying to generate solutions with a holistic approach is difficult. The graph visualization makes an iterative approach intuitive with the idea of expanding the graph. An algorithm starts at an arbitrary data element. It then finds all the possible nodes (called *available nodes*) that it can use as the next element in a candidate solution at that step. Then it selects one element from the set of available nodes and repeats the process until a solution is constructed. For a more detailed example of expanding graphs, please refer to Chapter 3 of *Artificial Intelligence: A Modern Approach* [5].

A basic type of search algorithm is an *uninformed search*. This type of search finds a solution with little to no information. An *exhaustive search*, which is an uninformed search, simply enumerates every candidate solution iteratively and then selects the best one (also known as a *brute force* approach). Since an exhaustive search checks every candidate solution to a problem, it is guaranteed to find the best solution. However, a search space can be too large to compute all the solutions in a reasonable amount of time. This leads to the idea of reducing the search space with smarter algorithms. By reducing the search space in intelligent ways, someone can still get a good solution while being able to solve the problem within a feasible time frame. This is the benefit behind using heuristic and metaheuristic functions.

2.3 Travelling Salesman Problem

A classic example of a computer science problem is the Travelling Salesman Problem (TSP). The problem is to find the shortest possible cycle through a graph of cities that visits each city exactly once. Let $G = (V,E)$ be a graph where V is a set of vertices and E is a set of edges. Then, the travelling salesman problem can be represented as G, where every element $v_1, v_2, .., v_n$ belonging to V represents a city in the problem and every element $e_1, e_2, ..., e_n$ belonging to E represents an edge between two of the cities. The graph has several properties. For example, it is weighted. This means that every edge belonging to E has some value x, that makes it comparable to other edges (in this case, the value is an integer denoting physical distance between the cities). Because it is weighted, the total distance of a cycle can be computed by summing all of the edges in a candidate solution. Also, this means a solution can be checked if it is the shortest cycle in linear time by comparing the computed solution cycle value against the shortest cycle value.

The graph is also strongly-connected. That means for every two vertices v_x and v_y belonging to V, there exists and edge e_{xy} belonging to E. Thus, you can get to any city from any other city in the problem. Another interesting property of the graph is the symmetry of the edges. A TSP can be *symmetric*, which means the edges are undirected. Let A (v_A) and B (v_B) be two cities in the problem. Then if the problem is symmetric, the distance to get from A to B (e_{AB}) is the same as the distance from B to A (e_{BA}). However, a TSP can also be *asymmetric*, in which the edges are directed. This

means the value of the edge from city A to city B differs from the value of the edge from B to A.

The TSP falls into the category of NP-Hard problems. As said before, a cycle can be checked if it is the shortest cycle in linear time given a solution path and the shortest path value. However, there are an extremely large number of possible cycles since the graph is strongly-connected. In fact, given the number of cities in the problem as *n*, there are exactly *n!* candidate solution cycles! Thus, every single cycle would have to be checked to guarantee finding the optimal solution. A basic brute force algorithm would have complexity O(*n!*), which means that for an input of just 100 cities you would need to check ~10^{158} cycles. Even with dynamic programming, where similar segments of cycles are stored in memory and are not recalculated for every path, you would have a complexity of O(c^n). This makes the problem infeasible to solve with a brute force search. Therefore reducing the search space for TSPs is crucial.

2.3.1 Nearest Neighbor List

The nearest neighbor list is a way to reduce computational and possibly memory overhead when computing a *tour* (or in other words, a cycle through the all the cities) in the TSP instance. The nearest neighbor list is obtained by sorting all the distances from one city to all the other cities in the problem. Then a subset of lowest cost cities, which we can denote by *nn*, is taken. This number is typically between 15

and 40 [6]. There are several tradeoffs for using the nearest neighbor list. For one, it can reduce the complexity of some of the steps from linear complexity (O(n)) to a constant complexity (O(1)). This is because only the cities in the nearest neighbor list are searched instead of all the cities in the graph. Only if all the nearest neighbors have been chosen must the rest of the graph be searched. This helps reduce computational overhead.

The use of a nearest neighbor list can also affect memory overhead. By only storing the costs of the nearest neighbors, and then computing other distances only when necessary, a large amount of memory overhead can be avoided. However, my approach does not use this method because it is concerned with optimal performance. An important note when using a nearest neighbor list is that the solution quality may degrade. Logically, an optimal tour will uses as many lowest cost edges (or the lowest cost edge available) as possible. In fact, many heuristic functions use this assumption as the basis for generating a solution. By taking a subset of lowest cost edges to search, the whole optimal tour or at least a majority of it is usually included in the search space. However, if the nearest neighbor list is too small, it can be impossible to find the optimal tour because it ends up lying outside the search space. Using a neighbor list is a heuristic, and there is no guarantee of optimality if one is used.

2.4 Heuristics

Heuristics are used for *informed search*, where informed guesses are made to reduce the search space. Let the search space be an arbitrary graph. As defined before, at each step of constructing a candidate solution there exist a set of available nodes. A heuristic function ranks the available nodes using problem specific information. So a heuristic function f(*n*) takes in an node *x* off the available node list and gives it an enumerable value. This information can then be used to reduce the search space to one candidate solution. While this solution is often not guaranteed to be optimal, heuristic functions often produce pretty good results. A big factor in how good a heuristic algorithm will be is the quality of the heuristic used and how good the rankings are. A poor heuristic may produce a solution that is far from optimal.

One very popular heuristic algorithm is the *greedy best-first choice*. A *greedy* algorithm is one that always makes the locally optimal choice. Choosing the heuristically best ranked node from the list of available nodes is called making the *locally* optimal choice, but it may not lead to the best overall solution to the problem, or making the *globally* optimal choice. After ranking the available nodes a greedy algorithm always chooses the node with the best ranking. An example of a greedy algorithm would be using the distances between cities in the TSP as a heuristic, and then always choosing the closest city as the next one in the tour. The idea of *best-first* means that the next node is chosen according to a specific rule. An example rule involves always selecting the node predicted to have the least cost to a solution if

chosen. Some best-first searches can involve backtracking and keeping track of past available nodes. A greedy best-first choice combines the two ideas: rank the currently available nodes by a heuristic that predicts the lowest cost to the final solution, and then always make the greedy choice and choose the best ranked node.

2.4.1 A Suboptimal Heuristic Solution

We now go through an example where a greedy search may not give the best global optimal solution because of local optima. Figures 1 and 2 on the next page depict an example problem.

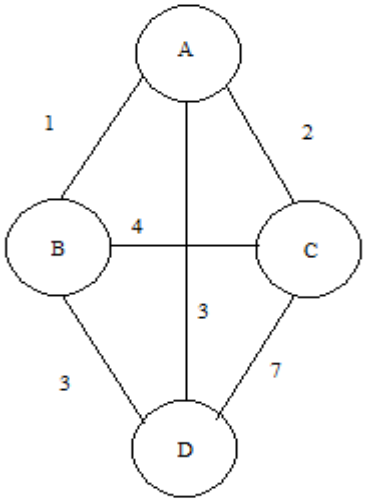

Figure 1: An example of a weighted graph.

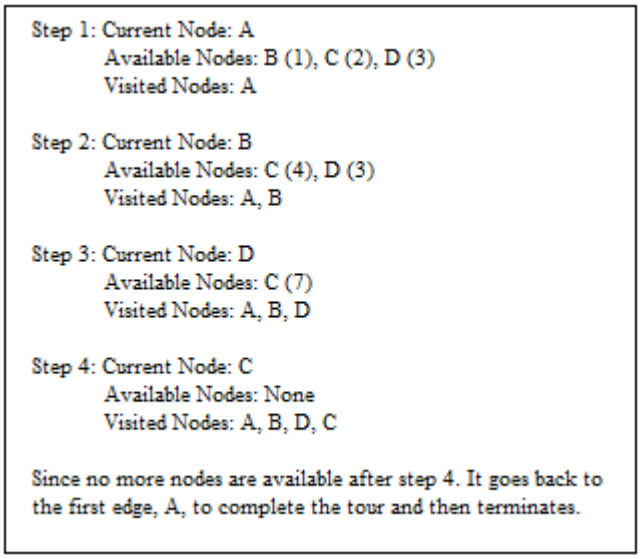

Figure 2: A step-through of a greedy algorithm on the graph.

Figure 1 denotes a simple problem graph. The graph above could easily represent a TSP graph, where each node is a city. The best solution is the cycle with the lowest cost. The heuristic used here is unimportant, other than the fact that it gives an enumerable value to each edge. When using a greedy search, the available edge with the lowest possible value is always chosen. Figure 2 steps through the choices the greedy search will make to create a cycle starting from node A. The final cycle created using the greedy search is A->B->D->C->A, with a total value of 13. However, the best cycle starting at A is A->C->B->D->A, with a total value of 12. This highlights a fundamental problem of many heuristic functions, the best choice made at each local node, may not be the best with respect to the global problem.

2.5 Metaheuristics

Metaheuristics are algorithms that use an underlying heuristic function and try to improve the solution quality. They attempt to diversify the search space and avoid converging to a poor solution because of local optima. Metaheuristics also iteratively try to improve their solution quality, which means that they do not terminate after just generating one solution. They will generate a number of candidate solutions, evaluate the respective solution qualities, and then use the information gained to improve the quality when generating new candidate solutions. There are many different types of metaheuristics, including but not limited to simulated annealing, particle swarm optimization, genetic algorithms, and ant colony optimization [34].

An important method that many metaheuristic searches use is the idea of *stochastic optimization*. This means that random variables are used when constructing solutions. This is a common tactic to diversify the search space. By adding randomness when choosing the elements of a solution, local optima can be avoided, and it is more likely a global optimum will be found. This is because the best value determined by the underlying heuristic is not always chosen. The Ant Colony Optimization algorithm is a metaheuristic that uses the idea of stochastic optimization to improve the solution quality.

2.6 Ant Colony Optimization

The Ant Colony Optimization (ACO) algorithm gets its name from real ants, because it models the way ants search for food. Ants in the real world begin by randomly searching for food. As ants find sources of food, they leave pheromone trails that allow other ants to find the food. Ants are influenced to travel along paths with pheromone trails. Over time, the closest food source will develop a stronger pheromone trail than other food sources. This is because ants can travel to the closest food location and back faster, depositing a stronger pheromone trail than other food sources. Since the trail is stronger, more ants will be influenced to travel along the path, further reinforcing the trail until the majority of ants converge to the trail. This is how ants find the closest food source. Pheromones also evaporate over time, which is important so ants do not continue to go to the same food source after it has disappeared. The ACO algorithm is a metaheuristic that models how real ants find food when generating a solution to a problem.

The ACO algorithm works in a similar way. It has two main stages: the construction stage and the pheromone update stage. There is also a daemon actions stage, which allows for statistical observation or adding additional heuristics, but this is not crucial to the algorithm. In the construction stage, individual ants construct a candidate solution to the problem using a probability function. The function uses a heuristic function and the amount of pheromone on the edges to decide which city to choose next. Once all ants have constructed their respective solution, the ants enter the

next stage. In the pheromone update stage, certain solutions (generally the best ones), deposit pheromones on the edges of their solutions. Also, old pheromone trails have their potency decreased during this stage to prevent early convergence to a suboptimal solution. After a certain time period, ants will converge to a near-optimal path through the graph, just like real ants converge onto a food source.

The following is pseudo-code give for the ACO algorithm:

```
Function ACO_metaheuristic
while(!termination_condition_met) {
    constructSolutions();
    daemonActions();
    pheromoneUpdate();
}
```

I will refer to one pass through this loop as an *iteration* of the ACO algorithm.

2.6.1 Termination Condition

The termination condition signifies when the algorithm should finish. Since the ACO algorithm attempts to improve on previous iterations, there are many termination conditions that could be applicable. In fact, for all metaheuristics, there is no general termination condition [6]. Therefore, there have been many different termination strategies developed. One possible termination condition is to bound the running time of the algorithm with a time limit, and finish after a certain amount of time has elapsed. Another possible condition is to terminate after a solution has been generated within a percentage deviation from a given optimum value. This approach requires knowledge of the problem before running the ACO algorithm, which may not always

be available. Another possible termination condition involves setting a bound for a maximum number of iterations completed without an improvement in solution quality. If a significant number of iterations complete without improvement to the best solution found, it is likely the algorithm has converged to a solution. All of these are possible termination conditions for algorithms attempting to solve a TSP.

2.6.2 Construction Stage

In this stage, each individual ant in the algorithm constructs a candidate solution to the problem. An ant is a problem specific data structure that can generate and retain the data of a completed candidate solution. The number of ants in an ACO algorithm run determines how many candidate solutions are constructed in every iteration. An important feature of the construction stage is that each individual solution can be constructed *independantly*, which means there is no communication between ants when they generate a solution. Ants are randomly placed at a starting node in the search graph, and then nodes are added to the candidate solution using a stochastic function until a complete tour has been generated. The stochastic function, is controlled by two variables, *alpha* and *beta*. Alpha is how heavily the pheromone trails are weighted in choosing the next node. Beta is how strong the orignal heuristic underlying the ACO algorithm is weighted in choosing the next node. For the function to work properly, it requires at least two matrices, one which has the heuristic values between nodes (the *heuristic matrix*) and one that stores the pheromone values

between nodes (the *pheromone matrix*). Now the stochastic function for applying ACO to a TSP can be generated. The heuristic matrix for a TSP stores the distances between cities. An ant *k* will rank the probability of choosing city *y* from current city *x* with the following function:

$$p_{xy}^{k} = \frac{\left(\tau_{xy}^{\alpha}\right)\left(\eta_{xy}^{\beta}\right)}{\sum_{y \in N}\left(\tau_{xy}^{\alpha}\right)\left(\eta_{xy}^{\beta}\right)}$$

In the above function, α denotes alpha, *N* denotes the set of unvisited cities for ant *k* and τ is the pheromone matrix. η_{xy} can be calculated a priori with the following equation: $\eta_{xy} = 1/d_{xy}$ where d_{xy} signifies the distance between cities x and y. The summation at the bottom sums the remaining probabilites of the non-visited cities. This is done so that the probabilities correspond to a traditional sample space. In other words $\sum_{y \in N} p(y) = 1$, where p(y) is the corresponding probability for visiting the city. Each ant ranks the remaining cities using this stochastic function, chooses one city by generating a random number between (0,1), and then repeats until all the cities in the TSP have been visited.

2.6.3 Pheromone Update

This stage updates the pheromone matrix using information generated from the construction stage. There are two stages to this stage, *evaporation* and *intensification*. In the evaporation stage, every pheromone edge is decremented by some value *rho*. Rho is always defined between (0,1). So every edge in the pheromone matrix, τ_{xy}, is

multiplied by rho to decrement its potency. Since early developed trails may be part of poor solutions, the process of evaporation allows these trails to be forgotten over time.

The next stage, intensification, lays down new pheromones in a manner determined by the pheromone updating system. There has been much research into how exactly the pheromones should be updated, including differences in tours to use, relative strength of the tours, and general application of pheromone parameters. Some examples of ACO systems include classic AS [7], MAX-MIN [9], Elitist [8], and Rank-Based [10].

We use the MAX-MIN system when developing the pheromone update stage on the GPU. The MAX-MIN system introduces two new parameters, τ_{max} and τ_{min}. The parameter τ_{max} is an upper bound on the amount of pheromone an edge can have. When the pheromone matrix is first created, all the values in the solution space are intialized with value τ_{max}. This allows for a more complete exploration of the search space, since initially all the edges are equally attractive with respect to their pheromone values. The value of τ_{min} is a lower bound on the amount of pheromone deposited on an edge. By forcing all edges to keep at least a minimum pheromone value, it keeps all the edges in the search space to help prevent them from not being explored in later iterations. The MAX-MIN algorithm only deposits pheromone on the best tour from each iteration. The value of the pheromones it deposits are quantitively derived from the tour's solution quality.

2.6.4 Daemon Actions

The part of the algorithm named "daemon actions" refers to all functions that are not neccessarily part of the traditional ACO algorithm. This step can occur before or after the pheromone updating stage. Some packages may want to output statistical information on the algorithm's performance during each iteration. Some ACO implementations may be used in conjunction with other optimization techniques, such as a local search [31]. They may achieve better results using such techniques. All these types of activities fall under the category of daemon actions. In the implementations, we disable the daemon actions section of the ACOTSP package since it is not critical to the execution of the algorithm.

2.7 Parallel ACO

The Max-Min ant system has complexity $O(m*n^2)$ for TSP, where m is the number of ants and n is the number of cities in the problem [9]. This is because the complexity of constructing each tour is approximately $O(n^2)$. This must be done m times since a tour must be constructed for every ant. This leads to a complexity of $O(m*n^2)$ for the construction stage. All the other functions in the algorithm have smaller computational requirements than the construction stage. Evaluating the values of the tours is complexity $O(m*n)$, since each ant must sum every element in its constructed tour to get the tour length. The pheromone evaporation function is complexity $O(n^2)$, since every edge in the pheromone matrix must be updated, and there are n^2 edges. The

pheromone intensification function is a linear time function with complexity O(n), there will be one computation for each edge in the best tour, and the number of edges is equal to the number of cities in the tour. However, all of the other functions are much less than the O($m*n^2$) complexity of the construction stage, and since Big O is an approximation, they are discarded in the overall complexity. Even though there seems to be a lot of work, all these functions put together are much less computation than a factorial O($n!$) number of computations. However, the performance can be improved. Let p denote a number of processors available to run the algorithm in parallel. With perfect scaling among processors a theoretical runtime of O($m/p*n^2$) can be achieved on each processor, if the work of constructing the tours is split evenly among the processors. Even taking into account the overhead from parallel computing, good results can still be achieved.

There have been many different methods of parallelizing ACO algorithms [17]. A set of ants sharing the same pheromone matrix and the same heuristic matrix is a *colony*. One way to parallelize the code is to split ants in the same colony up among several processors. Some ACO implementations, run multiple colonies with different parameters to diversify the search space even more. Another possibility involves decomposing the TSP domain. With this method, each processor p looks at generating a best path through a subset of cities, and then the segments are joined together into a tour. The theoretical complexity of the construction step on each processor would then be O($m*q^2$), where q is the cardinality of the largest subset of cities. It is impossible to

guarantee that the optimal solution is found with this type of decomposition, since edges are removed, reducing the search space. The hope is that it is still possible to find good solutions much quicker with this type of decomposition. The city distribution and the size of the subsets is a big factor in how effective this method may be. Acan [11] does an external memory approach which manages to get good results. The approach selects segments from already constructed good solutions, and then allows ants to construct the rest of the graph. However, constructing the rest of the graph after a segment is the same as searching a path in a subset of the cities for the TSP.

Chapter 3

GPU COMPUTING

Today, Central Processing Units, or CPUs, cannot handle all the computational requirements of modern day computer graphics. Almost all modern day computers have a Graphics Processing Unit (GPUs). GPUs are designed to have a large number of simple homogenous processors that can run many threads in parallel. This makes them a good choice for certain algorithms that can be phrased in a data-parallel fashion, where a GPU can yield a much higher price/performance than a comparably priced CPU. GPUs are also seen as useful for programs with high arithmetic intensity [36]. That means programs that perform large amounts of arithmetic operations in contrast to memory operations will especially benefit from GPU implementation. Before the Compute Unified Device Architecture [4], (CUDA) several researchers attempted to do general purpose GPU (GPGPU) programming using programming languages and toolkits that were designed for graphics, such as OpenGL [37]. However, with the advent of CUDA, programming GPUs to take advantage of their power for general-purpose applications has become much easier. This chapter explains development using CUDA and considerations for running on a modern day GPU architecture.

3.1 SIMT Model

CUDA programs have special functions, called *kernels*, that run on the GPU. Kernels are labeled with the __global__ keyword. They are called from the CPU side and are passed in two extra variables, the dimensions of the grid and the dimensions of the block. The grid variable is a one to three dimensional space that specifies the layout of the thread blocks, and the block variable is a one to three dimensional space that specifies the layout of threads within the blocks. For a more detailed account of this, it is recommended to read the NVIDIA CUDA Programming Guide [4]. The variability in dimensions creates a large possibility of possible indexing options for the programmer.

CUDA uses a Single Instruction Multiple Thread (SIMT) architecture, which is similar to the classic Single Instruction Multiple Data (SIMD) architecture [4]. The SIMD architecture involves concurrently performing the same arthimetic instructions across an array of data. Typically, this makes it useful for applications that perform many of the same arithmetic calculations on different data, such as graphics processing or particle simulation. SIMT differentiates itself from SIMD for two reasons. One, the programming is done from a different perspective: Instead of programming the entire width of the data, SIMT can program individual threads as well as coordinated threads. Also, when there is a divergence between threads, which means that threads are

performing different instructions on the data, the SIMT architecture will handle it automatically while it needs to be manually programmed in a SIMD model. However, divergence of threads is still something to be avoided for optimal performance.

CUDA executes instructions of threads in groups of 32 threads called warps. So when a thread block is run, it is split into warps that are processed on one of the symmetric multiprocessors on the card. Warps can be further split up into quarter or half warps for processing depending on the architecture of the GPU. Warps are always split up consecutively in the thread block, so the first warp in a block of 256 threads would be threads 0 to 31. How threads are scheduled is determined by the warp scheduler. Knowing how CUDA executes warps is important in creating efficient code and fully taking advantage of the GPU's potential. The performance of kernels can vary drastically depending on the divergence of threads or the coalescing of memory accesses by threads. Warps execute one instruction at a time, so all threads must be executing the same instruction for maximum performance. When a warp diverges due to a data-dependent conditional branch, the warp serially executes each branch path taken, disables all threads not on the divergent path, and then when all the paths are completed it converges the threads back onto the same execution path [26]. Therefore a large number of divergent branches in a kernel can significantly impact performance for the worse. The CUDA memory hierarchy is discussed in the next section.

With this type of architecture it may seem that CUDA is only suitable for data parallelism, where the same instructions on different data. However, with a deep

understanding of thread execution, it is possible to achieve more on the GPU, such as task parallelism [24]. Guevara et. al., manage to merge two different kernel tasks into one with proper indexing and memory strucuturing. This allowed for even increased performance by taking advantage of the massive amount of thread parallelism available. While it helps to think of CUDA from a SIMD perspective for performance reasons, CUDA maintains a high level of versatility in the types of applications that can be executed on a GPU.

3.2 GPU Memory

CUDA architectures use a shared memory model, which means that all threads in the program share access to the same physical data. This is in contrast to a distributed memory model, where each thread is given its own copy of the data. In shared memory, threads do not need to send each other data, since they all have access to the same global memory, which translates into a smaller communication overhead. In turn, shared memory models must be concerned with race conditions, as two threads should not attempt to read or write data that is currently being modified by another thread. Though CUDA does share a large global memory between threads, the global memory is at the end of a memory hierarchy in terms of access speed. Efficiently utilizing all levels of the memory hierarchy is crucial to achieving optimal performance.

The CUDA memory hierarchy contains several levels, shared among different levels of the GPU. All threads have access to a large global memory. When using

cudaMalloc(), the main function used to allocate memory on the GPU, it allocates memory from the GPU's global memory. All threads also have access to a special type of memory called constant memory. Values in constant memory are stored in global memory, but are cached for much faster access times. Constant memory is primarily for read-only variables that are accessed often, especially when all threads in a warp are reading the same address. Reads from constant memory are as fast as register reads in such a case. Finally, all threads have access to texture memory. Texture memory is also read-only and is optimized for certain data format reads. Texture memory is cached differently and has several properties differentiating itself from constant memory. In some cases, using texture memory is preferable to constant memory.

Besides global, constant, and texture memory, there is local memory, shared memory, and registers. Every thread has its own local memory. Local memory is not cached and is as expensive as accessing global memory. However, local memory accesses are always coalesced. Shared memory is on-chip memory on the multiprocessors. Since shared memory is per each multiprocessor, and each thread block executes on a multiprocessor, only threads within the same thread block can access the same shared memory locations. Registers generally have the best performance, but there are only a limited amount of them per a multiprocessor to be utilized. Shared memory and registers accesses have lower latency times than local memory accesses. Therefore, utilizing shared memory and registers is important for good performance. However, there is a limit on shared memory and registers per

thread. Using too much can limit multiprocessor occupancy, which may increase execution time. Grauer-Gray et. Al [25] conducted a study observing the effects of multiprocessor occupancy and the tradeoffs in such situations.

Threads should read memory in a coalesced manner. This means that all threads in a warp should access consective data elements in global memory. Shared memory is also more effective when accessed in a coalesced manner. Siegel et. al. [28] suggest allocating memory in a structure of arrays (SoA) rather than an array of structures (AoS), so related data is contiguous in memory. They show a 30-50% improvement of performance between such memory schemes. The difference between SoA and AoS is illustrated in the figure below, where x, y, and z are data elements in a structure. In the first array, the data is structured in a SoA while in the second it is structured in a AoS.

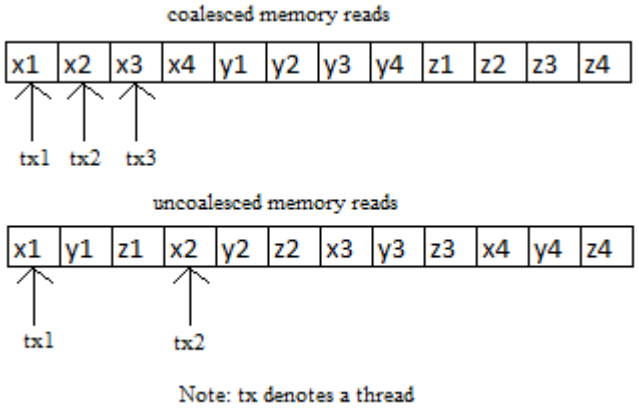

Figure 3: An example of coalesced versus uncoalesced memory access.

3.2.1 Memory Transfer

Another important aspect of memory coalescing is minimizing the amount of memory communication between the GPU and host. Since a memory transfer can take a significant amount of time, minimizing the amount of transfers is crucial to achieving good performance. For example, structures with pointers inside them cannot be transferred to and from the GPU as is, since pointers on the GPU cannot point into memory locations on the host device. It would be neccessary to individually copy every array from each struct to the GPU in such a case. This could lead to many memory copies per kernel run, which is highly inefficient. In my first CUDA implementation of ACO for the Travelling Salesman Problem, I solve this problem by consolidating the initial array allocation into one large array, and then having the structures point into the single array. Using this technique, the structures maintain the same functionality as the host side while minimizing memory transfers to one. The initial ACO implementation uses the technique of copying data between the host and the GPU, since it does not require an unmanageable amount of memory transfers. In my later implementation, I remove all memory transfers but one.

3.3 Synchronization Issues

Even coarse-grained algorithms, which are especially suitable for parallization because they require little communication between threads, may need to synchronize at certain points. The easiest way to handle this problem is to transfer the memory back

to the host and handle the synchronization with CPU code. However, this may not always be ideal, especially if this causes a large amount of memory transfers between the host and the GPU. CUDA suports synchronization only within thread blocks. This is accomplised with the use of barriers, where all threads will stop computing and wait at a barrier until all the threads in a block have reached the same point. Synchronization in between thread blocks can be done with the use of multiple kernel calls. In subsequent kernels, a parallel reduction technique can be used to merge data from other thread blocks.

3.4 CUDA for Evolutionary Computing

Many researchers have begun to realize the benefits of using CUDA for paralleling their applications and reducing execution time. In the field of evolutionary computing, which the ACO algorithm falls under, several algorithms have shown notable gains using GPUs. For example, Tsutsui and Fujimoto [13] showed a 3-12x speedup when using CUDA to parallelize a genetic algorithm. Franco, Krasnogor, and Bacardit [14] achieved an improvement of over 50x when applying CUDA to the BioHEL evolutionary learning system. These types of performance gains are good and make using GPUs an interesting option for these types of algorithms.

Chapter 4

INITIAL IMPLEMENTATION

My initial implementation of the CUDA ACO algorithm for TSP consisted of porting only the construction stage of the algorithm, because it is where the majority of the computation occurs. Every ant in the ACO algorithm (which each creates one tour) is run as a thread within a CUDA kernel.

4.1 Analyzing the Original Code

My initial parallel ACO implementation was done using CUDA and the ACOTSP package [1], a well-known ACO implementation. The ACOTSP package is a C implementation of the ACO algorithm used for solving Travelling Salesman Problems (TSP). The ACOTSP package was chosen because it is a highly-tuned sequential version of ACO and because it has many features, such as supporting multiple pheromone update functions and providing flexibility through command line options. It also provided a proven sequential version of code to compare my parallel implementation.

Before porting the code, the relative computation time of the ACOTSP's functions was analyzed using the gprof profiler [2]. It was found that the majority of

computation was spent in the construction phase of the algorithm, which took at least 80% of the execution time. After disabling functions to compute and output various non-essential statistics pertaining to the algorithm's performance, analysis of the results found that the construction phase took over 90% of the execution time. Therefore the construction phase of the package was ported to CUDA. This allowed for improved performance while maintaining the ability to use different pheromone update functions as desired. Speedups of up to 10x were achieved with my initial parallel implementation over the sequential one, just from performing the construction stage in parallel.

4.2 TSPLIB

The experiments in this paper use problems exclusively from TSPLIB, a widely known library of TSP problems [12]. The ACOTSP package supports parsing of the TSP files in this library. Another useful feature of TSPLIB is that the optimal solutions of the problems are documented within the library. Due to this being a popular source for TSPs, many other research papers explore the TSP instances defined within TSPLIB. Only strongly connected instances of TSP were chosen from the TSPLIB, though both symmetric and asymmetric TSPs were analyzed.

4.3 Issues in GPU Programming

This section describes some of the issues that arose when converting the sequential construction stage to run on the GPU.

4.3.1 Memory Coalescing

A big issue with porting to CUDA is having memory coalesced for transfers between the host and the GPU. Since a memory transfer can take a significant amount of time, minimizing the amount of transfers is crucial to getting good performance. The ACOTSP package uses a C structure called “ant_struct,” which contains two pointers to arrays. One array holds the solution constructed for each ant, the other array is a flag array to determine what cities have been visited during the construction step. Transferring these structures as is would require an unreasonable amount of transfers, since each pointer needs to be transferred individually. Structures with pointers inside them cannot be transferred to and from the GPU as is, since pointers on the GPU cannot point into memory locations on the host device. Many memory transfers per iteration would lead to poor performance.

To solve this problem, all fields in the structures were allocated as large consecutive arrays. Then, when the structures were created, a portion of this array was assigned to the pointer in the respective structure. Figure 4 below helps to illustrate this process. Then, the arrays can be copied in one memory transfer operation, and the proper sections can be referred to using offsets. This technique was also used for

arrays of floats. This allowed for efficient memory transfers while maintaining compatibility with the original functions. Note that if improper indexing occurs, incorrect results can be obtained with buffer overflows. However, bugs such as these would generally lead to segmentation faults as well, so these bugs are easy to identify and correct.

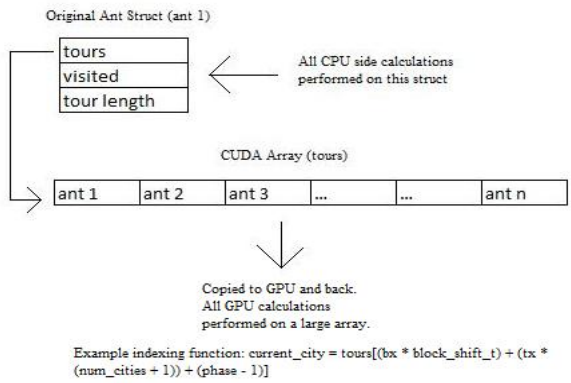

Figure 4: Shows how the memory from a structure may be mapped into a single array for use on a GPU.

4.3.2 Thread Divergence

An important issue with the parallel ACO algorithm is that it is impossible to program the kernel to guarantee thread convergence. This is because of the stochastic nature of the algorithm. Since each ant/thread will have a different partial solution in memory, the future cities it will be checking are different for each thread. Then when accessing the probabilities to access each city from the memory, there is no guarantee

those values are physically close in memory. One thread may be looking at one edge on the complete other side of the total matrix. Since the probability matrix is of size n^2, where n is the number of cities in the TSP, accessing the edges in different areas of the matrix will lead to thread divergence. Therefore, thread divergence is likely unavoidable between the threads.

4.3.3 Random Number Generation

A key part of the ACO algorithm is the usage of a Random Number Generator (RNG) to select cities in the construction step. Random number generation in CUDA cannot have threads modifying the same seed to avoid race conditions. Also, every thread must have an original seed, otherwise each thread will calculate the same values. This is because if given the same starting seed a RNG will generate the same sequence of numbers. A good random number generator must also have a good distribution of values within the bounds given.

The original ACOTSP package RNG used a common seed, which was not a problem with sequential code but did not work with a parallel implementation. For my initial implementation, a Linear Congruential Generator with a Combined Tausworthe Generator (as described in GPU Gems 3) was used [16]. When the research began, there were not many RNG packages for GPU computing. In the second and multi-GPU implementations, I used NVIDIA's recently developed RNG package [15].

4.4 Abstraction of Implementation

My program uses parallelism at level of each individual ant. That is, each ant is given its own thread to run on a stream processor. Each stream processor is part of a multiprocessor that executes a thread block. For performance reasons, I ran thread blocks of 256 threads (and therefore 256 ants). The probability matrix is copied to the GPU before each iteration is run. Each ant constructs its own solution to the TSP problem on the GPU, and then the GPU returns all the tours to the CPU for completion of the algorithm. Figure 5 shows the high-level structure of the CUDA version of the algorithm, which is similar to the ACO pseudo-code shown earlier. Before the main loop, structures and arrays are allocated to hold the pheromones and tours constructed by each individual ant. The loop in Figure 5 repeats until the time limit set for the algorithm has expired. My implementation uses time as the termination condition.

```
Initialize_GPU_memory_and_constants();
  while( !termination_condition() ) {
    copy_data_to_GPU();
    // executes construction stage for each ant in colony
    runTSP();
    copy_data_from_GPU();
    daemon_actions();
    //  any supported pheromone update function
    //  supported by the ACOTSP package
   pheromone_update();
}
```

Figure 5: Pseudo-code of the algorithm when running the construction stage on the GPU.

4.5 Platform

The experiments were performed using three different GPUs, the TeslaC870 GPU, TeslaC2050 GPU, and the Tesla C1060 GPU. The Tesla line of cards is NVIDIA's first offering of general-purpose computing GPUs. The C870 is the first offering, which only supports CUDA compute capability 1.0 and has 128 stream processors split across 16 multiprocessors with 1.5GB of memory. The C1060 has 240 stream processors split among 30 multiprocessors and 4GB of memory. The C2050 is based on NVIDIA's new Fermi architecture, and has 448 stream processors among 14 multiprocessors and 3GB of memory. It supports compute capability 2.0. I use Xeon processors to execute the original sequential version of the ACOTSP package. Results were normalized using the Intel Xeon E5335 2.0Ghz processor paired with 2GB memory for the C870, and a Xeon E5530 2.4Ghz processor with 24GB of memory for the C2050 and C1060.

4.6 Experiments and Results

I varied two variables in our experiments: problem size and the size of the ant colony (i.e., the number of ants used to find tours). For each of these combinations, ten trials was performed on each platform. While the original ACOTSP package has support for several different types of pheromone update functions, I used only the max-min variation for my experiments. Even though my version works with the other types of pheromone update functions, I found the difference in speed and solution

quality was consistent between the different pheromone functions. For the ACO parameters, Alpha was set to 1, Beta was set to 2, and Rho was set to 0.5. The program had a max memory usage of 1GB. For my graphs, I test each problem/ant combination for a timed interval, and then average the results. Speedups were calculated by enumerating the number of iterations per second and then comparing the execution times for the same number of iterations between the sequential and parallel versions.

Figure 6 shows my results for the symmetric problems. The results show a definite improvement in execution time when using our CUDA implementation over the sequential CPU implementation on large sized graphs and a high number of ants. As I increase the problem size the performance gap between the CPU implementation and the CUDA implementation also increases. Since the construction step in the ACO algorithm is general and applicable to many pheromone update functions, these speedups are achievable for several variations of the ACO algorithm.

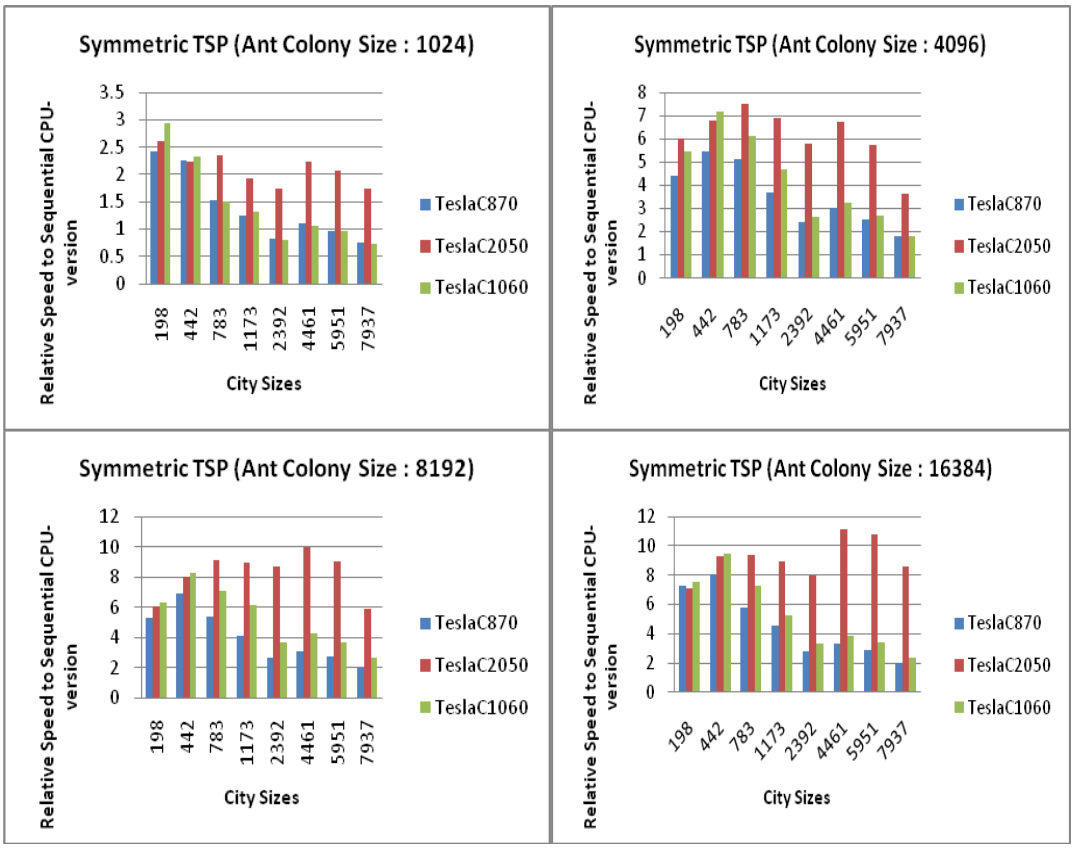

Figure 6: Speedup gained from running the initial implementation over the sequential algorithm on different problem sizes. Each graph represents a different number of ants.

4.7 Asymmetric TSP Results

The original ACOTSP package supported only symmetric TSP graphs. However, there are many asymmetric graph problems in TSPLIB. The package was then modified to support running on asymmetric TSPs. First the parser was changed to

support TSPLIB's file format for asymmetric problems. Different functions that assumed symmetry in the program's matrices, such as pheromone evaporation, were modified to support asymmetric problems. The results achieved for asymmetric graphs are posted below in Figure 7. I used the same methodology for experiments as the symmetric graphs.

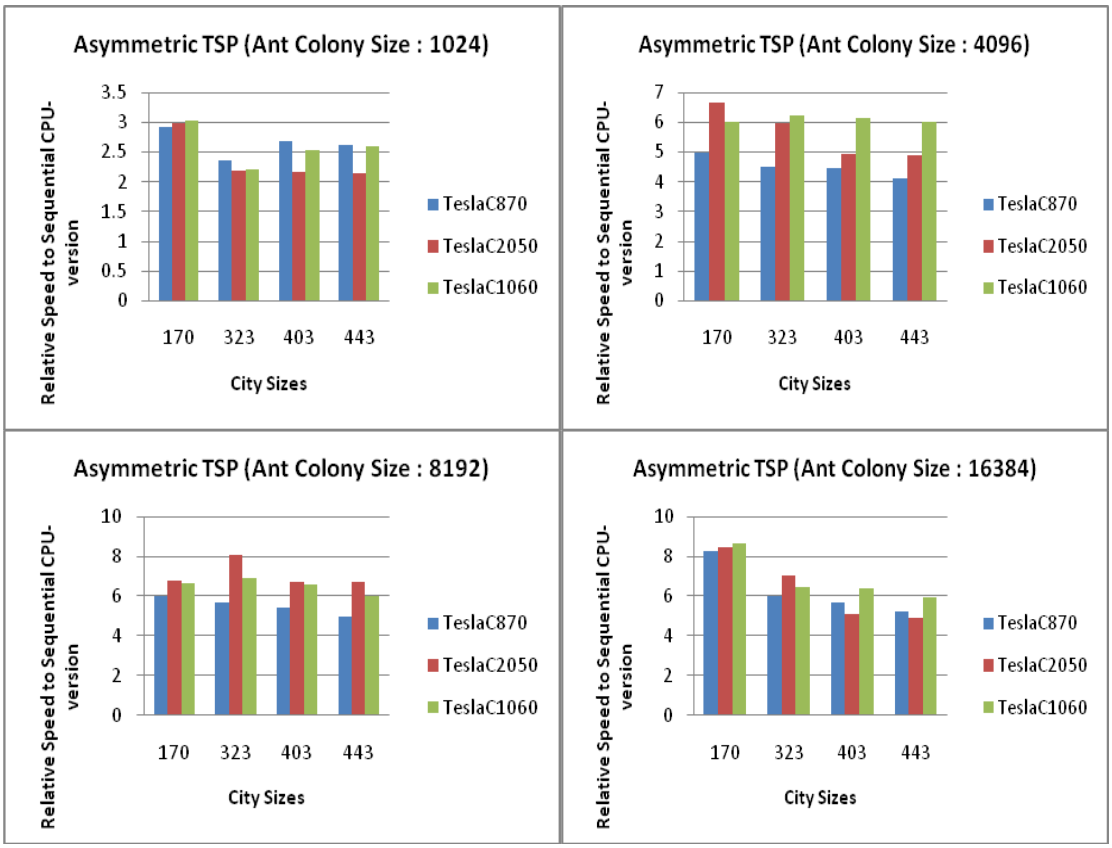

Figure 7: Speedup gained from running the initial implementation over the sequential algorithm on asymmetric problems. Each graph represents a different number of ants.

4.8 Analysis of Results

I highlight some of the results show in Figures 6 and 7. For example, the C2050 does not always outperform the older cards, especially at smaller problem sizes of under 1000 cities. A possibility for this is that the extra computational power of the newer card is not being fully utilized at these sizes. However, for large city sizes, e.g., over 1000, the C2050 gets significant performance improvements over the sequential code, and its speedups are much greater than the other two cards. For symmetric problems and the smaller ant colony size of 1024, our GPUs only achieve a performance improvement of up to 3x. For larger city sizes and older GPU models, I see a slowdown over the sequential version. I am currently investigating this anomaly. However, for symmetric problems and ant colony sizes above 1024, I consistently see good speedups especially for the newer GPUs based on the Fermi architecture. I can achieve speedups of almost 11x compared to a highly-tuned sequential version of ACO. For all the ATSP problem sizes, the speedups are relatively similar between the three cards. Overall, the newer card Fermi-based GPU performs better than the older GPUs. A possible reason the Fermi-based GPUs may achieve such better performance at larger city sizes is they may handle thread divergence better. The Fermi cards have a new memory architecture that works better with divergent memory accesses between threads. For larger problem sizes, it becomes much more likely that threads will become divergent between memory accesses because there is a much larger memory

space, and each thread can be accessing information anywhere in the probability matrix.

Chapter 5

SECOND IMPLEMENTATION

By porting the construction step to a GPU, there was definitely an improvement in performance. However, it is possible that even better speedup can be achieved. If an estimate that only 10% of the execution time is done sequentially outside the construction stage, the execution time is bounded by a maximum 10x speedup if Amdahl's Law is applied. While the sequential execution time is really between 5-10%, the rest of the algorithm can still be optimized for a significant overall speedup. By porting the rest of the algorithm to the GPU there is a two-fold benefit. First of all, the parallelism in the remaining functions in the algorithm can be further exploited. For example, each ant can calculate its own tour value in parallel. Secondly, all but one memory transfer between the GPU and CPU can be eliminated. Only the best tour would ever need to be transferred back and forth. Both these factors should increase the overall performance of the ACO algorithm.

5.1 Calculating the Best Tour

One part of the ACO algorithm that could be ported is the ability to find the best tour. To find the overall best tour on the GPU, we can have each ant must evaluate its

own tour value in parallel. This involves running a kernel that simply sums the distances values in each tour. The next step is to somehow compare these tour values in parallel to calculate the best one. To do this a parallel reduction step is necessary. This is somewhat tricky to do in our case for two reasons. First, indexing for the tour values must be preserved to be able to link it to the corresponding tour it represents. Secondly, there is no thread synchronization between blocks, which means multiple kernels must be used.

To solve the problem of index preservation, a second array that holds the original index values of the tour values is used. Then when two tour values in the parallel reduction step are swapped, the original index values are swapped as well. In this manner the index of the tour the value corresponds to can be tracked. To solve the problem of thread synchronization, the reduction step first runs only between threads in each block. After that kernel completes, another kernel is executed that performs a reduction step on the previous kernel's results to find the best overall tour.

5.2 Pheromone Updating

The MAX-MIN update system was used on the GPU for updating the pheromones in each iteration; this algorithm uses the best ant from each iteration to update the values. Stutlze [9] achieves good performance using this system and shows it is a good update strategy.

When evaporating pheromones and maintaining trail bounds on the GPU,

k blocks of size of 32 are run, where *k* is the number of cities to the next power of 2 divided by 32. A global id is then generated by multiplying the block id times 32 plus the thread id within the block. Threads with global ids greater than the number of cities are ignored. Each thread updates *n* edges, where *n* is equal to the number of cities. For the global update function we again run block sizes of 32, and *k* blocks. In this function, every edge updates one edge along the best path.

5.3 Second Implementation Results

Figure 8 shows the speedup over the CPU implementation when running the whole algorithm on the C2050 GPU for a number TSP problems and utilizing a variety of ants counts ranging from 256 to 8192. Using 256 ants is interesting because at this number only one thread block is running on the GPU for the construction stage. This means only one multiproccesor is executing on the GPU, yet the performance achieved is very similar to the performance of the CPU version.

The results show a definite improvement in execution time when using a CUDA implementation over the sequential CPU implementation on large sized input sets and when utilizing a large number of ants; the trial with the largest speedup of almost 16X uses the fnl4461 TSP city set as input with 8192 ants. The results show that when increasing the problem size, the performance gap between the CPU implementation and the CUDA implementation also increases. Also, performance may be limited by divergence between the threads. Due to the stochastic nature of the algorithm, threads

running in the same warp may be reading from completely different parts of the probability matrix.

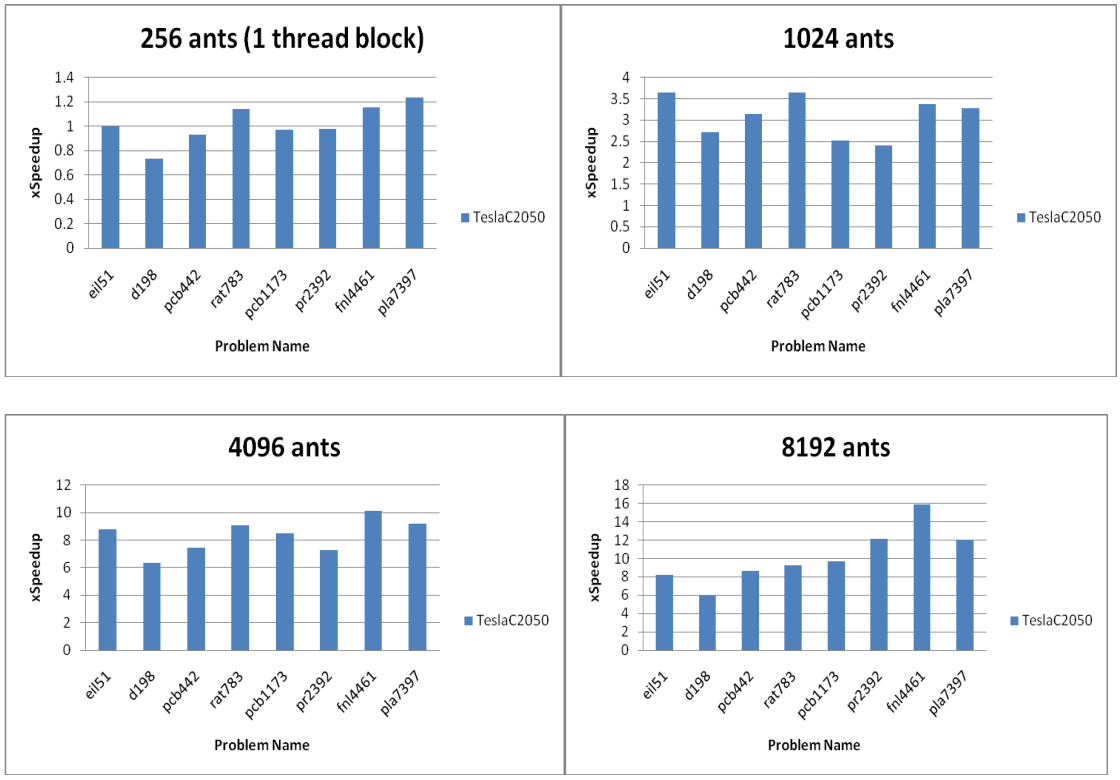

Figure 8: Speedup gained from the second implementation over the sequential implementation. Each graph represents a different number of ants.

5.4 Solution Quality

Solution quality was tested to make sure the implementations were providing good solutions. Solution quality between the first and second implementation was similar.

For testing solution quality, ten trials were run and capped the execution time at 5 minutes per a trial. Values of alpha = 1, beta = 2, and rho = 0.5 were used. Each trial ran with 1024 ants in testing the quality of the solutions. To compute how accurate the experimental results were, RPD, or the relative percent deviation was used. RPD is calculated using the following formula:

$$RPD = 100\ x\ (\frac{cost_{actual} - cost_{best}}{cost_{best}})$$

The $cost_{actual}$ is the optimal tour value for the problem and the $cost_{best}$ is the tour value found by the ACO algorithm. The ACO algorithm was run for four different problems and the following results were found:

Problem	Max RPD	Average RPD
eil51	0	0.469
d198	2.402	2.959
pcb442	3.761	4.498
rat783	6.62	7.031

Table 1: A table showing the max and average RPD results for the second implementation on different problem sizes.

Both the max RPD and the average RPD are considered. The max RPD signifies the RPD for the best solution found out of all the trials. The average RPD is done using the average of the solutions found from all our trials for that problem. The results are

consistent with the RPD achieved in both the sequential and the initial implementations for the same amount of iterations.

Chapter 6

MULTI-GPU IMPLEMENTATION

A possible improvement to my original implementations was a multi-GPU implementation. The idea behind a multi-GPU approach is to split up the work among several GPUs and then combine the results together. The initial implementation I developed of a multi-GPU approach was to split the work up of the ants. This meant that each GPU would run its own collection of ants. The main idea here is to see if the program benefits from further parallelism among ants. This approach could also be easily extended to a multi-colony approach. This would require giving each GPU kernel its own pheromone matrix. Then problem parameters, such as alpha, beta, rho, the number of neighbors, etc. could be different for each of the GPUs.

Another approach to a multi-GPU implementation would be to split up the domain space. This would mean each GPU would create segments of a candidate solution and work on a subset of the cities. This could improve execution performance, since each GPU would be splitting the work. This would also allow execution of bigger problem sizes. Note, after reaching a TSP size of about 8000 cities, a 1GB GPU would runs of memory. However, by splitting the cities up into subsets, and finding a path for each subset on the GPU, bigger problem sizes may be possible. This is a big factor, since

bigger problems are where performance is more of an issue. This type of decomposition may have a negative effect on the solution quality, since it reduces the search space by not searching edges that exist between the subsets of cities.

6.1 Threading

When running a multi-GPU program, each kernel call must be on a separate CPU thread. This is because every time a kernel function is called, the thread calling it will block until the kernel terminates. Therefore multiple CPU threads are required to run multiple kernel calls. The CPU threading is OS dependent, but all that is needed is the ability to run CPU threads and then wait for them to terminate collectively. To distinguish between multiple GPUs, every GPU installed on a machine is designated by an integer value. There exist CUDA functions that allow for managing which GPU to execute the kernels on.

When creating a multi-GPU implementation, every GPU needs its own device pointers for its problem dependent memory allocations. We handle this by encapsulating all the memory requirements for a kernel into a structure. In an initialization stage, determine the problem information each GPU will need and create the appropriate structure. Then pass the appropriate structure to each thread, let each thread initialize the GPU it will execute the kernel on, and then allocate the memory with the structure. Then each GPU should be able to run the correct kernel call.

6.2 Multi-GPU Results

We implemented a multi-GPU version that splits up both the ants and the cities. The algorithm evenly splits the ants evenly among the GPUs. It also naively splits the cities among the GPUs based on their sequential location in the input file. So if there are *n* cities and *x* amount of GPUs, the first *n/x* cities in the input file are put on the first GPU, the next *n/x* on the second GPU, and so on until all the cities have been assigned to a GPU. The best paths within each subset of cities are then connected to form the whole tour after the algorithm runs. The existence of an edge between the subsets can be guaranteed because the graph is strongly connected, allowing for the construction of a complete tour. Scott Grauer-Gray, a graduate student at University of Delaware, assisted in the coding and development of the multi-GPU implementation. Scott developed the majority of the code that splits the work along the cities. This implementation should have the benefit of increased execution time and allow for testing of larger problem sizes.

In our experiments, we run each trial for two minutes. The same values of alpha, beta, and rho are used as in our previous experiments (1, 2, and 0.5 respectively). The GPUs used were all Tesla C2050s. The following execution time results were obtained:

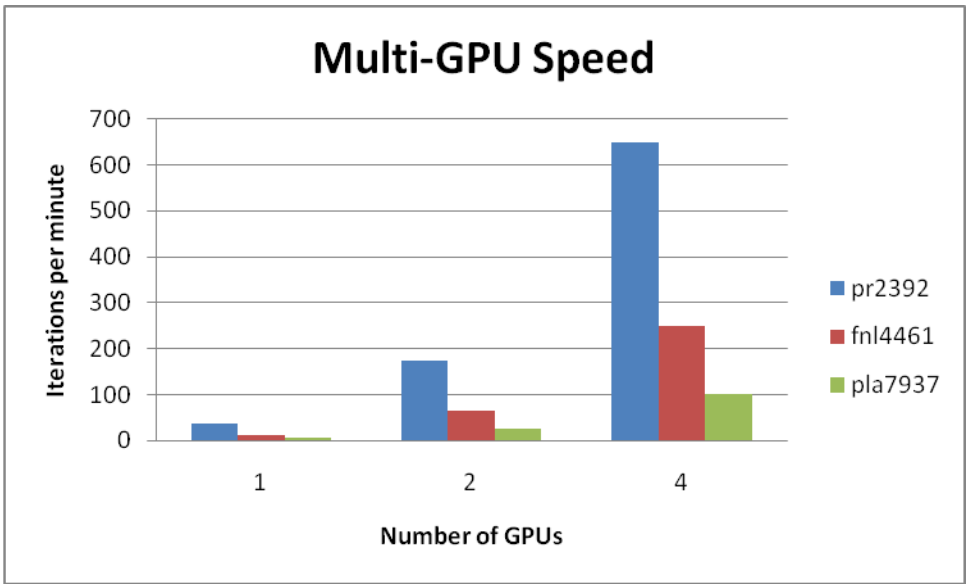

Figure 9: A graph showing the number of iterations per a minute the multi-GPU version runs at for different number of GPUs.

The 1 GPU test speed is similar to the results in the previous single GPU implementation. The speedup results show an approximate speedup of 4x every time we doubled the number of GPUs. This is because there is a twofold benefit from the optimization – one doubling for splitting the cities, another for splitting the ants. However, solution quality must be analyzed here, because we reduce the solution space by creating subsets of cities, possibly removing good solutions. RPD was computed as described before and these results were achieved:

# GPUs	pr2392	fnl4461	pla7937
1	15.5	17.5	17.9
2	15.8	16.6	38.5
4	22.6	17.9	44.1

Table 2: Shows the RPD for the multi-GPU implementation on a trial run of different problems.

There results were achieved after only one trial and as such requires much more testing before being conclusive. These results show degradation in the solution quality with our multi-GPU ACO algorithm. The gaps in final solution quality between the GPUs may increase as the algorithm is run for longer periods of time.

6.3 Improving the Multi-GPU Implementation

This implementation, while using a naïve split and therefore suffering losses in solution quality, lays a framework for future work. More elaborate domain decomposition can be performed using a modified external memory approach, and splitting the cities up as segments of an already found good solution. This builds upon work done in previous external memory implementations as described in section 2.5 [11]. This implementation also supports running different parameters and using a multi-colony approach such as done by Bai et al. [21]. Also, Weiss [29], who implements an ACO algorithm for data mining on GPUs, mentions a multi-GPU approach as part of his future work section. This section is mainly to show that a

multi-GPU approach can give even further performance improvement, especially in cases where a large number of ants are needed or the domain can be split in an intelligent way.

Chapter 7

RELATED AND FUTURE WORK

This chapter highlights other work published related to my research and plans for how to proceed with future research on the topic.

7.1 Related Work

There have been promising results of parallel ACO implementations for specific problems. For TSP, Randall and Lewis [18] developed a parallel version on a MIMD (Multiple Instruction Multiple Data) machine with MPI (Message Passing Interface). They proposed that the communication was slowing the implementation down and it would have benefited from a shared memory model over a distributed one. Shared memory model implementations have been developed using OpenMP for several different applications of ACO. For example, Delisle et al. [19] implemented an OpenMP ACO algorithm to solve an industrial scheduling problem. Their implementation provided a 2x speedup with 2 processors and up to a 5x speedup with 16 processors. Bui et al. [20] proposed a parallel ACO algorithm for the maximum clique problem. Their results ranged from a 40% performance increase on two

processors and up to 6x faster with an eight-core system. Both papers show the potential for a SIMD shared memory implementation of ACO algorithms.

Recently, there have been a few papers on solving the TSP problem using CUDA-implementations of ACO. Bai et al. [21] developed an implementation using MMAS (Max-Min Ant System). They achieve a 2.3x speedup, even though they performed a much larger workload on the GPU than the sequential version. Fu et al. [22] published a recent paper using ACO to solve TSP problems. They achieve significant results by improving an ACO MATLAB implementation on TSP problems. They used the Jacket toolbox to improve over the sequential MATLAB implementation. Finally, the AntMiner system [29] implements a parallel ACO algorithm for a data mining problem on GPUs using CUDA. Weiss achieves very good results in his implementation, gaining up to 100x performance in some cases. The fact that he achieves great results applying ACO to another problem on GPUs shows that many other problems may benefit from this type of platform/algorithm combination.

7.2 Future Work

My analysis of the ACO algorithm on the TSPs looks at improving performance by exploiting parallelism inherent in the algorithm. While I achieve a significant increase, performance could still be improved. A more complex domain decomposition as described in Chapter 6 could be another research possibility. However, another possible avenue of research is applying parallel ACO algorithms to other NP-Hard

problems. In the related work above, there were several papers that did just this and achieved good results. One possibility I looked at was applying a parallel ACO algorithm to the problem of instruction scheduling.

7.2.1 ACO Instruction Scheduling

If a search algorithm can be used for solving the instruction scheduling problem, that means someone can probably apply ACO to solving instruction scheduling. The ACO algorithm could use the dependency time between instructions as the heuristic part of the probability equation. Then the instructions currently available for execution would be the local nodes used for picking the next element in the partial schedule. Pheromones would be laid down on the dependency edges between the instructions.

There are several challenges to implementing parallel ACO for instruction scheduling. The algorithm would need to incorporate validating which edges are possible to select between iterations. The ACO algorithm would need to be given the amount of computing resources in the instruction pipeline. Also, evaluation of the instruction set could be a challenge. Different architectures may have different processing times for instructions, so the best way to evaluate the solutions would be an actual execution of the reordered instructions. If a large number of solutions was generated, this could be a very costly operation.

There have already been several ACO implementations of instruction scheduling done using CUDA. One implementation done by Wang et al. [23] manages to achieve better performance using a hybrid MAX-MIN ACO algorithm rather than just list scheduling. ACO has also been done for other types of scheduling problems. Socha et al. [38] manages to get good performance for scheduling university courses using an MAX-MIN ACO algorithm as well.

Chapter 8

CONCLUSION

The research shows promising results. I achieve up to 10x speedup when porting the construction stage of the algorithm, and up to 16x speedup when porting the whole ACO algorithm while maintaining comparable solution quality per an iteration. The multi-GPU version achieves an additional speedup of 4x every time the number of GPUs is doubled, though solution quality suffers. These types of speedups are especially relevant to larger problem sizes, where each iteration of the ACO algorithm takes a large amount of time.

I also provide a strong framework for future work in the topic. I proposed several ways for improving the multi-GPU implementation, and laid some groundwork for applying the ACO algorithm to the Instruction Scheduling problem. Further optimization may also be possible in the CUDA kernel. The results also suggest that other metaheuristic algorithms may also benefit from GPU implementations as well.

BIBLIOGRAPHY

[1] Thomas Stuetzle. *ACOTSP, Version 1.0*. 2004.

[2] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.* 39, 4 (April 2004), 49-57.

[3] M. Dorigo and T. Stützle, *The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances*. Handbook of Metaheuristics, 2002.

[4] *NVIDIA CUDA Programming Guide 3.1*. 2010.

[5] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education.

[6] Marco Dorigo and Thomas Stützle. 2004. *Ant Colony Optimization*. Bradford Co., Scituate, MA, USA.

[7] M. Dorigo and L.M. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation, 1*, 1, (1997), 53-66.

[8] M.Dorigo, V. Maniezzo and A. Colorni. Ant System: Optimization by a colony of cooperating agents, *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, 26(1), 29-41, 1996.

[9] T. Stützle and H. H. Hoos, MAX-MIN Ant System. Future Generation Computer Systems. 16(8):889--914,2000.

[10] B. Bullnheimer, R. F. Hartl and C. Strauss, A New Rank Based Version of the Ant System: A Computational Study. Central European Journal for Operations Research and Economics, 7(1):25-38, 1999.

[11] A. Acan, “An external memory implementation in ant colony optimization,” in *Fourth International Workshop on Ant Colony Optimization and Swarm Intelligence – ANTS 2004, Lecture Notes in Computer Science*, 2004, pp. 73-82.

[12] G. Reinelt. TSPLIB—A traveling salesman problem library. *ORSA J. Comput. 3,* (1991), 376-384.

[13] S. Tsutsui and N. Fujimoto. 2009. Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers* (GECCO '09). ACM, New York, NY, USA, 2523-2530.

[14] María A. Franco, Natalio Krasnogor, and Jaume Bacardit. 2010. Speeding up the evaluation of evolutionary learning systems using GPGPUs. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (GECCO '10).

[15] *CUDA Toolkit 3.2 Math Library Performance.* 2011.

[16] L. Howes and D. Thomas. Efficient Random Number Generation and Application Using CUDA. In *GPU Gems 3*, Hubert Nguyen, chapter 37, Addison Wesley.

[17] S. Janson, D. Merkle, and M. Middendorf. In *Parallel Metaheuristics*, chapter Parallel Ant Colony Algorithms, John Wiley & Sons, 2005, 171-201.

[18] Marcus Randall and Andrew Lewis. 2002. A parallel implementation of ant colony optimization. *J. Parallel Distrib. Comput.* 62, 9 (September 2002), 1421-1432.

[19] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné. Parallel Implementation of an Ant Colony Optimization Metaheuristic with OpenMP. In *International conference of parallel architectures and complication techniques (PACT), Proceedings of the third European workshop on OpenMP (EWOMP 2001)*, pages 8-12, Barcelona, Spain, September 8-9 2001.

[20] Thang N. Bui, ThanhVu Nguyen, and Joseph R. Rizzo, Jr.. 2009. Parallel shared memory strategies for ant-based optimization algorithms. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (GECCO '09). ACM, New York, NY, USA, 1-8.

[21] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu. 2009. MAX-MIN Ant System on GPU with CUDA. In *Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control* (ICICIC '09). IEEE Computer Society, Washington, DC, USA, 801-804.

[22] Jie Fu, Lin Lei, and Guohua Zhou. A parallel Ant Colony Optimization algorithm with GPU-acceleration based on All-In-Roulette selection. In *Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on*, 260-264.

[23] Gang Wang, Wenrui Gong, and Ryan Kastner. 2005. Instruction scheduling using *MAX-MIN* ant system optimization. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI* (GLSVLSI '05). ACM, New York, NY, USA, 44-49.

[24] Marisabel Guevara, Chris Gregg, Kim Hazelwood, Kevin Skadron. "Enabling Task Parallelism in the CUDA Scheduler," in *Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA)*. Raleigh, NC. September 2009, pages 69-76.

[25] S. Grauer-Gray, J. Cavazos. Optimizing and Auto-tuning Belief Propagation on the GPU. In The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2010) .

[26] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40-53.

[27] Abid M. Malik, Tyrel Russell, Michael Chase, and Peter Beek. 2008. Learning heuristics for basic block instruction scheduling. *Journal of Heuristics* 14, 6 (December 2008), 549-569.

[28] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. 2009. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops* (ICPPW '09). IEEE Computer Society, Washington, DC, USA, 174-181.

[29] R.M.Weiss, Gpu-accelerated ant colony optimization, in: W.M.W. Hwu (Ed.), GPU Computing Gems: Emerald Edition, Applications of GPU Computing, Morgan Kaufmann, 2011, pp. 325–340.

[30] Eliot Moss, Paul Utgoff, John Cavazos, Carla Brodley, David Scheeff, Doina Precup, and Darko Stefanovic;. 1998. Learning to schedule straight-line code. In

Proceedings of the 1997 conference on Advances in neural information processing systems 10 (NIPS '97), Michael I. Jordan, Michael J. Kearns, and Sara A. Solla (Eds.). MIT Press, Cambridge, MA, USA, 929-935.

[31] The Traveling Salesman Problem: A Case Study in Local Optimization, D. S. Johnson and L. A. McGeoch, *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (editors), John Wiley and Sons, Ltd., 1997, pp. 215-310.

[32] Russell, Tyrell. Learning Instruction Scheduling Heuristics from Optimal Data. "Master's thesis", University of Waterloo. 2006.

[33] Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing.

[34] Sean Luke, 2009, *Essentials of Metaheuristics*, Lulu, available for free at http://cs.gmu.edu/~sean/book/metaheuristics/

[35] 2001. *Introduction to Algorithms* (2nd ed.). MIT Press, Cambridge, MA, USA.

[36] Mark Harris. Mapping Computational Concepts to GPUs. In *GPU Gems 2*, Hubert Nguyen, chapter 31, Addison Wesley.

[37] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999

[38] Krzysztof Socha, Joshua Knowles, and Michael Sampels. 2002. A MAX-MIN Ant System for the University Course Timetabling Problem. In *Proceedings of the*

Third International Workshop on Ant Algorithms (ANTS '02), Marco Dorigo, Gianni Di Caro, and Michael Sampels (Eds.). Springer-Verlag, London, UK, 1-13.