Ruslan Kudubayev
Churchill College

Computer Science Tripos PartII Project Dissertation

Parallelising the Travelling Salesman Problem on NVIDIA®CUDA$^{\text{TM}}$architecture

May 12, 2010

Proforma

Name:	**Ruslan Kudubayev**
College:	**Churchill College**
Project Title:	**Parallelising the Travelling Salesman Problem on NVIDIA®CUDA™architecture**
Examination:	**PartII Computer Science Tripos, June 2010**
Word Count:	**Approximately 10000**
Project Originator:	Ruslan Kudubayev
Supervisor:	Dr Jagdish Modi

Original Aims of the Project

The main aim of the project is to study parallelisation characteristics of the Travelling Salesman Problem using NVIDIA CUDA. The project should present a good overview of approaches that were taken by other people and justify that the approach that was chosen is suitable for parallelisation. The secondary aim is to show how GPU can outperform CPU by demonstrating the speedup gained. It is also desirable to study scalability characteristics of the implementation as it performs on different CUDA-enabled devices.

Work Completed

The GPU implementation on CUDA was carefully prepared and presented. The CPU implementation on Java was also completed and thoroughly documented. Both implementations were driven through experiments in order to obtain trends that were analysed in the Evaluation.

Special Difficulties

None.

Declaration

I, Ruslan Kudubayev of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Acknowledgements

I'm very grateful to:

Dr Anton Lokhmotov, who assisted me in choosing the topic for my dissertation,

Dr Jagdish Modi, who supervised my project,

Dr John Fawcett, who provided generous feedback and advice,

and Mr Bernard Breton, who kindly provided additional hardware for testing purposes.

Chapter 1

Introduction

The aim of this project is to investigate parallelisation approaches for the Travelling Salesman Problem on modern GPU devices to demonstrate their promising capabilities. The main objective is to compare GPU performance to the corresponding CPU performance. The algorithm chosen is Ant Colony Optimisation which is an approximation algorithm aimed at problems like the Travelling Salesman Problem.

1.1 Motivation

Massively parallel computation has for a long time been considered expensive and was often not affordable to ordinary people interested in performing extensive dataset computation. In present days our society is faced with a fast-paced market of supercomputing, so the general trend in the industry shows that there is high performance where there is a larger market of consumers. For example, in the old days of computing, people used to write efficient programs that were performed on laser printers, because at the time they were more powerful in certain tasks than mainstream CPUs.

One such example we are observing today is the emergence of Graphical Processing Units (GPU) produced by various vendors. These devices are capable of performing massively parallel computation that achieves phenomenal performance for tasks that involve a lot of calculations such as scene rendering.

There is no doubt that GPUs are capable of producing smooth and realistic graphics. However what is becoming more prevalent now is general-purpose computation on GPU (GPGPU), which allows very expensive real-world computation tasks to be tackled on a relatively cheap graphics hardware. Particularly interesting developments are in scientific applications using GPUs, such as n-body and Monte-Carlo simulations.

When looking for a practical non-trivial problem that is very interesting to tackle on a GPU, the Travelling Salesman Problem(TSP) was learned from the Complexity Theory course. The main practical applications of this problem are mainly in logistics, but it also appears in planning, manufacturing of microchips, DNA sequencing and many more. This problem has for a long time been one of the most challenging optimisation problems in the algorithmic community, and this is still the case. Furthermore TSP is known to be NP-Complete [6], so solving it is generally very hard unless a polynomial solution is found at some point in future. If it was possible to solve this problem efficiently, then there would be many interesting applications, because this problem is connected to all other NP-Complete problems(CLIQUE, IND-SET, KNAPSACK, etc). Currently there are very good heuristic-based solvers that can solve instances of TSP to optimality or approximate answer to a reasonable amount.

The central motivation for this project came from a desire to study parallelisation aspects of the Travelling Salesman Problem, which is something that could potentially be very useful for practical purposes. It was decided that NVIDIA's CUDA-enabled series of graphic cards constitute a suitable platform that is suitable for demonstrating those parallelisation aspects.

1.2 Reading Guide

The following lecture materials were used in this study:

- Comparative Architectures
- Complexity Theory
- Algorithms I and II
- Software Engineering
- Concurrent Systems and Applications
- Advanced Graphics

The rest of this work is divided into four chapters. Chapter 2 discusses preparation taken prior to the implementation phase. It explains some key facts about hardware that should be understood, and also explains a significant part of the theoretical background behind this project. Chapter 3 gives an overview of how the implementation was approached. Chapter 4 focuses on evaluating the work that was done and derives some key results that were achieved. Finally, Chapter 5 concludes the study by summarising what was achieved and analysing the success of the whole project.

Chapter 2

Preparation

2.1 Hardware

The main hardware used for this project is an NVIDIA Geforce 9400M that comes with any modern MacBook laptop. The way this device operates is by using programmable shaders to process the data stored in texture memories for producing 3D graphics. Many industry vendors support different ways of exploiting these devices for graphics, usually using a language like OpenGL or Direct3D. One can of course choose to write massively parallel high-performance programs on OpenGL by overwriting shader functions. This approach is called General Purpose GPU (GPGPU) programming. GPGPU is quite challenging as an approach overall and is being abandoned as modern technology like CUDA and OpenCL continue to emerge.

The overall structure of a typical program that uses a GPU to accelerate some parallelisable subtasks is shown on Figure 2.1.

Most NVIDIA devices support the architecture called CUDA which provides a simplistic abstraction of GPU hardware. CUDA is a variant of C, which uses standard parallel programming paradigms: the device-side computation-carrying units are threads. One can synchronise threads to certain "barriers" in a program, and one can also group threads to allow communication between them. CUDA does not provide locking, but there are atomic operations that can be used in any way to achieve lock-like behaviour using semaphores. So the only tools for parallel decomposition are threads, grids (thread groupings) and synchronisation. This of course limits the range of problems that can be tackled on CUDA architecture.

The memory hierarchy in CUDA has a number of layers, but only three levels were mainly used in this project:

- GPU device memory is on the lowest layer of the hierarchy. It is device memory that is available to the device in total (typically 256MB or 512MB on most common cards).

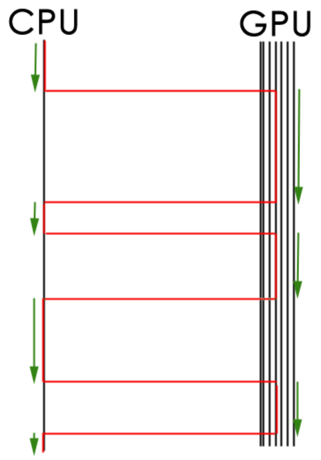

Figure 2.1: Interleaving computation between CPU and GPU synchronously. It is also possible to run code asynchronously.

s

This memory is designated for general use by OpenGL in graphics mode: it stores texture information as well as shape(polygon) coordinates in that memory. Global memory can also be shared between processes. A typical penalty for accessing device memory is about 100 clock cycles. Generally all GPU memories support gather and scatter operations for simultaneous use of same memory locations (see Figure 2.2).

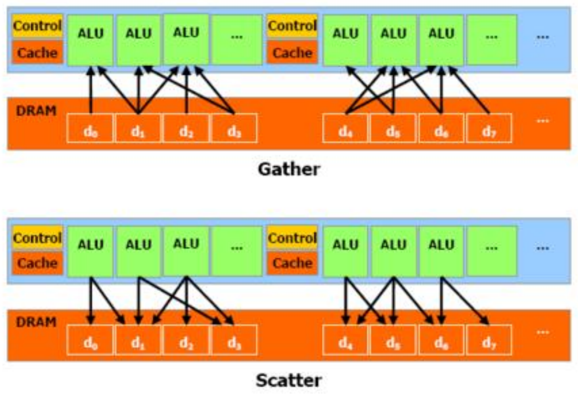

Figure 2.2: GPU memory is capable of gather and scatter operations (image taken from the NVIDIA website)

- Shared memory is in the middle of the memory hierarchy. In graphics mode this usually corresponds to texture buffers that are used for accelerated rendering. A typical size of shared memory is 64KB and penalty for accessing it is around 16 clock cycles on most devices. Due to design limitations, shared memory can be used between 512 threads only (maximum grid block size).

- Register files and local memory are at the highest level that threads operate when they execute instructions. The access time is normally just a few clock cycles.

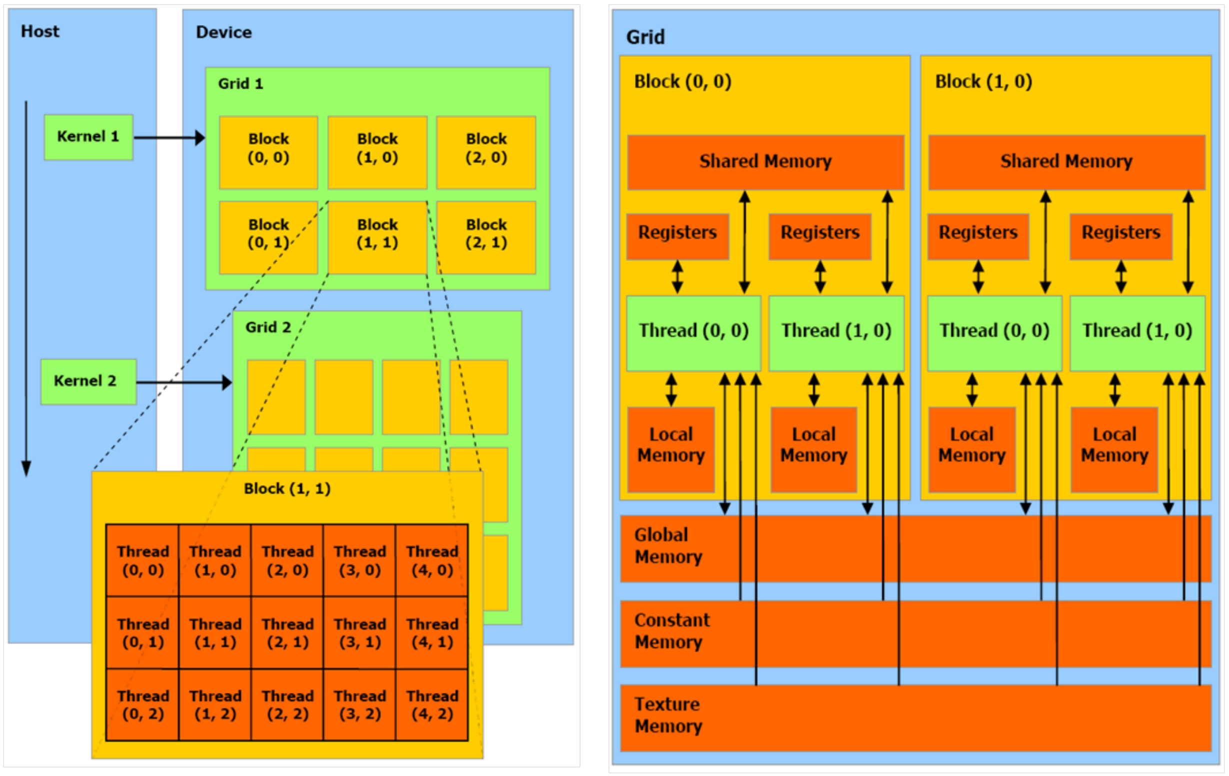

Figure 2.3: CUDA grids and memory hierarchy (this image is combined from images on the NVIDIA website)

One important feature of all CUDA-powered NVIDIA GPUs is that all threads are hardware-scheduled. The device uses so-called warps of threads (typically size 8) and schedules threads dynamically on the fly. The scheduler also hides penalty of accessing device memory by executing other threads while one is blocked waiting for memory feedback.

NVIDIA Geforce 9400M has 16 CUDA cores and memory bandwidth of 21GB/sec. The full technical specification of the device can be found on the NVIDIA website.

2.2 Theoretical background

Firstly, define a Hamiltonian cycle:

Given a directed graph G, a Hamiltonian cycle is a path that visits every vertex exactly once and comes back to the starting vertex.

An informal definition of the Travelling Salesman Problem that is widely accepted follows:

> Given a weighed directed graph G, find a Hamiltonian cycle of minimum total weight.

More formally, define a distance matrix D (where $d_{i,j} \in \mathbb{N}$) of weights of all edges in the graph $G\langle$E,V$\rangle$. The solver is seeking to find a matrix U, where $u_{i,j} \in (0,1)$ and $\forall i \in V.(\exists! j \in V.i \neq j \wedge u_{i,j} = 1) \wedge (\exists! k \in V.i \neq k \wedge u_{k,i} = 1)$ (this is just to ensure that there is only one incoming and one outgoing edge at each vertex), such that $\sum_{i,j \in V} u_{i,j} d_{i,j}$ is minimum. Note that the sum would be infinity in the case where there is no Hamiltonian cycle in the graph G.

One can clearly see that, in total, there are $O(N!)$ paths in every problem instance (simply taking permutations of all vertexes). This means that the lower bound for TSP is around $O(NlogN)$. However, no approaches have been found so far that can solve any classical TSP problem in polynomial time. The problem is clearly in class NP (non-deterministic polynomial), and moreover it can be shown that this problem is also NP-hard [3, 6]. Hence this problem lies in the NP-Complete set of problems and can be trivially reduced to the SAT (boolean formula satisfactiability) problem which is known to be ultimately NP-hard according to the Cook's theorem.

The CUDA environment is similar to theoretical Parallel Random Access Machine(PRAM) environment. One can reason about parallelisation perspectives of TSP by examining that the best complexity class for parallelisation is NC (Nick's Class) problems [10]. P-Complete problems are so far known to be hard to parallelise, however it is still not known whether NC=P, in which case there would be ways of parallelising P-Complete problems efficiently. Therefore given that it is unlikely that P=NP, NP-Complete problems could also be inherently not suitable for parallelisation.

2.3 Approaching TSP

There are many ways of tackling this problem. The most successful ones are very complex, such as *Concorde TSP Solver* [2]. It is based on a branch-and-cut approach and can only handle symmetric problem instances. However, one can easily turn any asymmetric TSP instance into a symmetric one, just by doubling the number of vertices and doing some distance matrix manipulations [7].

It is desirable that any solution to TSP can handle any graph topology equally. This is driven by the fact that it is easy to come up with heuristics that perform extremely well on certain types of graphs, while fail on the others.

Most solutions fall into two categories: exact and approximation solutions.

2.3.1 Exact solution

One can solve some instances of TSP by simply brute-forcing all possible paths in time $O(N!)$. This approach is parallelisable by dividing the search space into subspaces for each parallel processor. Such subdivision is subject to scalability and also ensuring that subspaces do not overlap.

Another approach is to use dynamic programming. Held and Karp [9] proposed a simple solution that is $O(N^2 2^N)$.

The most effective approach is branch-and-bound which is a tree search technique that uses a bound heuristic which narrows down the search tree. Such exact solution method can also be coupled with Lin-Kernighan heuristic [11] to obtain an efficient exact TSP solver and also a good approximating solver that finds local minima.

An important thing to note is that any kind of branch-and-bound technique would involve a tree search, which is tricky in a parallel environment because of inherent control dependencies. Pekny and Miller [12] developed a parallel branch-and-bound algorithm for TSP on a data-flow computer which is not exactly similar to the target architecture for the work. The exact solution approach is therefore not examined in this study. Another reason is that CUDA is not capable of creating threads dynamically, in contrast to Java-like multithreading environment. Harish and Narayanan [8] studied various tree searching techniques with CUDA and their report shows unimpressive results.

2.3.2 Approximation approach

The main algorithm used in this study is called *Ant-Colony Optimisation*(ACO) which was originally developed by Marco Dorigo [5]. The idea is inspired by how real ants find food in nature. Every ant lays some amount of pheromone everywhere it passes. Ants move by sensing pheromone and cognitively preferring to head towards places where there is more pheromone. As a result, there is an efficient network of trails that ants use to signal where the food is.

In ACO for TSP, virtual ants move from vertex to vertex until they complete a Hamilton cycle. Their total distance covered is then calculated and pheromone values are updated on each edge of the covered path accordingly. Virtual ants use the following probabilistic function to randomly choose where to go next. This is called the *state-transition rule*:

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\delta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\delta_{i,j}^{\beta})} \tag{2.1}$$

where

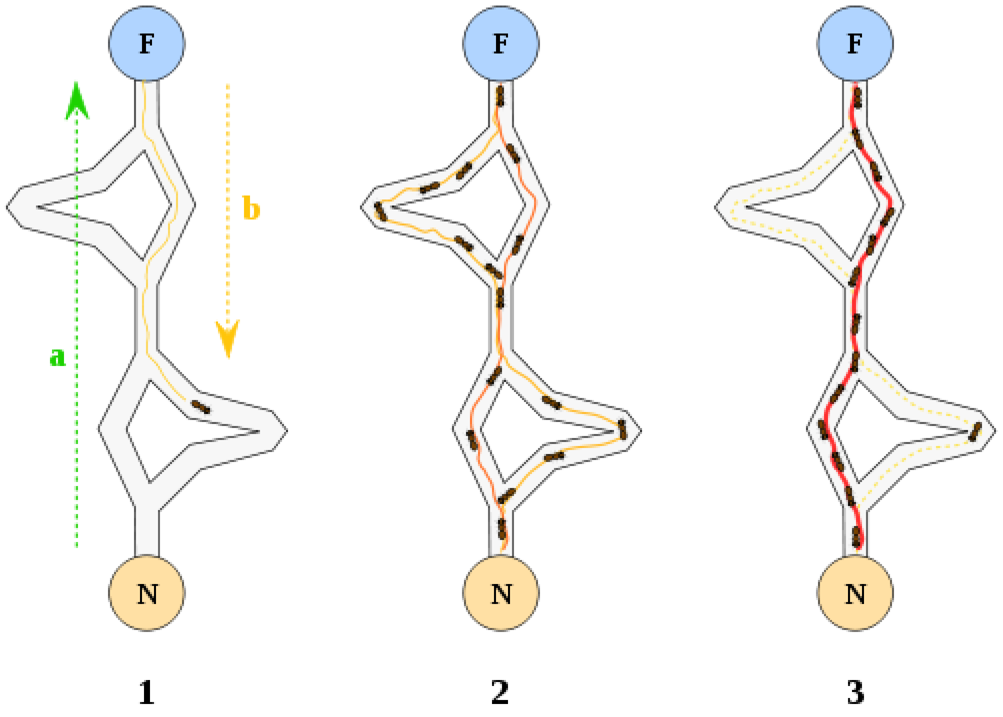

Figure 2.4: Ant-Colony pheromone convergence (image taken from wikipedia)

- $\tau_{i,j}$ is the amount of pheromone on edge (i, j).
- $\delta_{i,j}$ is the desirability of edge (i, j) $(= 1/d_{i,j}$ to make shorter edges more desirable).
- α and β are parameters to control the influence of $\tau_{i,j}$ and $\delta_{i,j}$ respectively. These were set to 1 in this study because of problems related to losing floating point number precision. However, it would also be interesting to study how these parameters affect the operation of the algorithm in a separate study.

Pheromone update is normally done using the following formula (applied by each ant as it visits edges of the graph). This is also called the *local update rule*:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \tau_0 \tag{2.2}$$

where

- ρ is the local rate of pheromone evaporation. Set to be 0.1 in this work, but it would be interesting to have a separate study on this one as well.
- τ_0 is an approximation pheromone characteristic. Controls how much pheromone is laid.

The ACO is run for all virtual ants until they complete their trips. One iteration may not be enough, so several iterations of ACO are performed on the same graph and same τ data until a satisfying global solution is found. Between iterations an *ACS* (Ant Colony System) optimisation is applied in which best path found so far gets modified [5]. The *global update rule* needs to be able to detect which edges belong to the best solution of preceding ACO iterations.

$$\tau_{i,j} = (1-\gamma)\tau_{i,j} + isonbest(i,j) \times (\gamma/\delta_{iterationbest}) \tag{2.3}$$

where

- γ is the global rate of pheromone evaporation, which is set to be 0.1 again for this study.
- $isonbest(i,j)$ is equal to 1 if the edge (i,j) is on the best path of the most recent iteration, and 0 otherwise.
- $\delta_{iterationbest}$ is the total weight of the best Hamiltonian cycle found on the most recent iteration.

The global update rule aims to perform pheromone evaporation after each iteration. It also aims to add more pheromone on the most recent best path in order to increase the probability of that path being picked up again. Other reported approaches in ACO tried to use the best global solution so far instead of per-iteration result. This might be good, but may discriminate against other paths too aggressively, therefore a more relaxed alternative was chosen.

The ACO algorithm may resemble some features of a non-homogeneous Markov random process based on discrete time. Interestingly, Dorigo and Stutzle [4] showed that ACO is guaranteed to find an optimal solution with a probability that can be made arbitrarily close to 1 if given enough time. They also proved that after a fixed number of iterations has elapsed since the optimal solution was first found, the pheromone trails on the connections of the optimal solution are larger than those on any other connection. The approach taken in this work uses more heuristics in order to decrease the convergence time, therefore it might not strictly be eligible for the fact proved by Dorigo and Stutzle.

The ACO approach is inherently good for parallelisation because ants are completely independent of each other. Ants only informally communicate through updating pheromones in the τ matrix, but because of the random nature of the algorithm, communication does not need to be synchronised. The algorithm is also suitable for CUDA, because the number of ants does not have to be dynamic and not many parallel decomposition tools are needed (such as locks, etc).

2.4 Test data

Preparing a problem instance generator may be an easy task to do, but the task of checking whether the result is actually the optimal tour is NP-Complete being just a decision problem version of TSP. Luckily, there is a package called TSPLIB [1] which combines asymmetric TSP instances with answers. The correctness of answers provided in the library is of course questionable, but it is the tool that is used for most of the research that is being done on TSP.

2.5 Requirements Specification

Given an instance of a TSP problem the program (both CPU and GPU versions) will attempt to produce an answer within certain bounds and report the time taken to reach it.

The fact that ACO is still an approximation algorithm imposes that the algorithm has to have a success criteria, which was chosen to be within 20% to the optimal result. So, the implementation should record the time it takes to reach 20% accuracy and also record the time to reach optimality, if the optimal result is eventually produced.

It was decided that the implementation should allow many parameters that can be changed. That is driven by the fact that a lot of coefficients should be derived experimentally in order to extract better performance. For a good evaluation it is also needed to be able to change parameters like the number of ants, number of ACO iterations, etc.

2.6 Organising the development process

It was decided to first tackle the GPU implementation in order to get a feeling of how complicated the CPU version should be in order to be able to compare them fairly. In order to accelerate working with CUDA files, a number of specialised bash scripts were written. Those scripts would facilitate any repetitive work such as copying test instance files from the TSPLIB library, launching the solver and then recording results.

For the CPU implementation the Java language was chosen, developing using Eclipse IDE. Java also provides great support for multi-threading. The language is generally slower than C at runtime, but that fact was considered insignificant when compared to the ease of coding and cross-platformness.

All files must be backed-up regularly, so all project files were placed in a github repository[1] and therefore also made open-source.

[1] `http://github.com/guruslan/tsp_antcolony`

Chapter 3

Implementation

3.1 GPU implementation

The GPU implementation contains a few files that are divided into two parts: host side and device side.

Functionalities in files are divided as follows:

- `tsp_antcolony.cu`: Main entry to the program. Contains the host code which is executed by the CPU which then launches the GPU code.
- `tsp_antcolony_kernel.cu`: GPU side of the program which performs execution on the device.
- `tsp_antcolony.h`: A header file containing some constants for array sizes, etc.
- `MersenneTwister_kernel.cu`: GPU-side random number generation routine. This file is an external dependency and comes with the CUDA SDK.
- `MersenneTwister.h`: A header file for the random number generator.
- `dci.h`: Another header file for the random number generator.
- `runtsp.sh`: A bash script that automates file handling when executing batches of tests together.

3.1.1 Host code

The host code is mainly located in the `tsp_antcolony.cu` file. In the main `runTest()` method of that file, there is a device setup routine which is just calling some inbuilt functions like `cudaSetDevice(cutGetMaxGflopsDeviceId())`. This setup routine ensures that a CUDA-enabled device is present in the system and will alert if no such device exists. Following the setup routine, there is also a random number generator(RNG) setup routine, about which there is more in Section 3.1.3 below.

What the program does then is it parses the input file which is in TSBLIB format and puts all data in local arrays allocated on the heap. This move is justifiable because all problem instances that are met in this study are well within the memory bounds of most computers. The parsing is achieved by the `getSize` and `getGraph` subroutines which are trivial in implementation.

Another part of the initialisation routine is the host copying data to the GPU. This is done by calling special functions like `cudaMalloc` and `cudaMemcpy` for allocating memory on the device and then copying data of appropriate size from the host to the device. The data that are allocated and copied include: distances array(`delta`), pheromones array(`tau`), random numbers array(`R`, more about it in Section 3.1.3), problem size integer(`N`), result integer(`best`), result path vector(`best_path`).

At the point when all the data has finished copying, the device is ready to run the core part of the program. As shown on Figure 3.1, the host side should simply launch the ACO routine (or *kernel*, in CUDA terms) on GPU for `K` number of times. Here, `K` can be any value that satisfies the needs of the study as it is only significant for cases when the algorithm is failing to find the right answer, so increasing `K` will give some extra time. The value for `K` that was used for this study is `2500`.

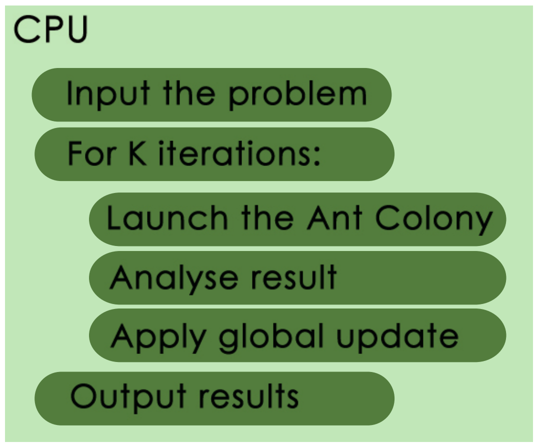

Figure 3.1: High-level implementation of the CPU side of the algorithm

Calling the ACO kernel (which is located in `tsp_antcolony_kernel.cu` and defined as external function at the host side):

```
colonise<<< blocks, BLOCK_SIZE >>>(d_C, d_A,
d_Rand, d_best, d_P, n, damping, tau0);
```

where

- `colonise` is the name of the kernel that is defined in `tsp_antcolony_kernel.cu`. See Section 3.1.4
- `BLOCK_SIZE` is a constant that reflects hardware limitations that force blocks to contain at most 512 threads. In this project, it is set to `32` which is also the minimum number of threads per block. The choice of `BLOCK_SIZE` is considered insignificant for the results obtained.
- `n` is the number of vertices in the given graph
- `blocks` is the number of blocks that the kernel would need. It is pre-calculated by `blocks = iDivUp(ANTS, BLOCK_SIZE)`. Here
  - `iDivUp` performs division with rounding up.
  - `ANTS` is a constant that specifies how many ants there are in total in the system. The intention is to be able to control the `ANTS` constant in order to discover the optimum number of ants to be present in the system.
- `d_C` is a pointer to the τ pheromone matrix in the device memory
- `d_A` is a pointer to the δ distance matrix in the device memory
- `d_Rand` is a pointer to the matrix of random numbers in the device memory. See Section 3.1.3 for details.
- `d_best` is a pointer to the integer in the device memory that stores the best result of the iteration. This is set to infinity at the start of each iteration.
- `d_P` is a pointer to the vector in the device memory that stores the best path that corresponds to `d_best`.
- `damping` is a constant that corresponds to the local rate of pheromone evaporation, ρ, defined in Formula 2.2. ρ could be derived experimentally, but in order to save time, value `0.1` was used, following recommendations in Dorigo's book [5].
- `tau0` is an approximation pheromone characteristic, τ_0, which is also defined in Formula 2.2. More about choosing this characteristic in Section 3.1.2.

In between iterations, the best answer so far is copied from the device to the host and the host compares the result to the actual answer. If it is the case that 20% accuracy has been achieved, the program would record the time taken to reach this result and if the optimal answer is not achieved, will continue iterations. If the optimal answer is reached, the program will record the time taken and will exit the iterations loop. At the end of each iteration, the ACS heuristic is applied, which is simply applying the global update rule defined in Formula 2.3. Instead of copying the `tau` matrix from the device and performing the update on the host, the global update rule is applied on the device.

This is achieved by launching another kernel which is specialised for doing that particular update. The kernel approach will save time and bandwidth on copying data. The key idea is that different kernels are working on the same data.

```
update_pheromones<<< grid2, threads2 >>>(d_C, d_best, d_P,
n, damping);
```

where

- The parameters are same as defined above. Notice that `damping` is reused here. This corresponds to another fact taken from Dorigo's book, which is that $\gamma = 0.1$ [5] and it is also equal to ρ.
- `grid2` and `threads2` are pre-calculated grid descriptors of size that matches the size of the `tau` matrix.

  ```
  // dimensions of the global update kernel
  dim3 threads2(BLOCK_SIDE_UPDATER, BLOCK_SIDE_UPDATER);
  int side_blocks = n/BLOCK_SIDE_UPDATER;
  if (n%BLOCK_SIDE_UPDATER != 0) side_blocks++;
  dim3 grid2(side_blocks, side_blocks);
  ```

  `BLOCK_SIDE_UPDATER` constant is arbitrarily set to 16, which makes 256 threads per square block. The matrix is set to be equipartitioned into square blocks in order to designate one thread per matrix element. The difference between this grid and the one used for `colonise` is that this one uses square blocks to work with the matrix and `colonise` grid uses vector blocks to work with ants. Both approaches are interchangeable and considered not significant for performance. They are just ways of parallel decomposition.

After the program has finished all ACO iterations the results are displayed for the user to the output stream. The data included in the report has both timer recordings for approximation time and optimality convergence(if reached). The program also processes the graphs to output them in *DOT* format. The helper subroutine that does the formatting is called `outputfordotformat`. It is used twice to produce two files: `original_graph.dot` and `tau_graph.dot`. Those files can then be fed into *Graphviz* open-source program which in turn produces graph visualisations that can help to observe the final state of the system more easily.

3.1.2 Choosing τ_0

The value τ_0 influences the amount of pheromone added by each ant as it performs the local update rule. If the value for τ_0 is chosen to be too low, the convergence rate may be too slow. If the value is too high, there is a risk of overshooting in the beginning and a bad route would get too much preference which in turn may result into slower convergence rate or even not result into anything having a cyclic sequence of states.

It turned out in early stages of development that setting τ_0 to a constant (like `1.0`) has varying effects across problem instances. Dorigo and Stutzle again recommend that it is best if τ_0 is as close to $\delta_{optimal}/n$ as possible (average edge weight of an optimal tour)[5]. *NearestNeighbour* heuristic can get close to the optimal answer and it is very easy to implement. It operates by performing a route by choosing the closest neighbouring vertex at each step until a solution is obtained. The computational complexity of calculating τ_0 is $O(n^2)$, but it is only calculated once.

3.1.3 Note about RNG

The algorithm heavily relies on random numbers. In particular, it uses random numbers to make a probabilistic choice using the state-transition rule (Formula 2.1). Another use of random numbers is to choose whether to explore properly or simply use the best pheromone edge (more about this in Section 3.1.4).

It turned out that CUDA does not support random number generation in the same way as most CPUs do. The reasons are clear - no shader would ever need to generate random numbers, so there is simply no hardware for doing that. Another reason is of course that there are so many threads running in parallel, it is hard to drive different random numbers to different threads on same clock cycle (standard SIMD architecture problem).

The approach that was developed first used same formulas as ANSI C's `rand()` function does only applied with different starting cell values for each thread. Note that `rand()`'s algorithm is very simple, it works by simply applying a linear function to a previous value that is stored in its internal cell. It quickly turned out that this approach is not suitable because random numbers generated were still dependent on each other. As a result, the Ant Colony didn't converge for larger problem instances.

Following advice from project overseers, all random numbers that were needed by the kernel are preloaded into the device memory rather than being generated on the fly. This approach proved to be more successful and was strengthened by using *Mersenne-Twister* method for parallel random number generation on CUDA. The algorithm itself is complicated, but NVIDIA provides the source code for it in the CUDA SDK[13].

The NVIDIA's Mersenne-Twister kernel (`MersenneTwister_kernel.cu`) was adapted to this project to generate $(2 \times n - 1) \times ANTS$ random numbers for each iteration (each ant needs two random numbers on each of $n - 1$ steps and another random number to determine the starting vertex). This required some initialisation code which was inserted in the main initialisation section of the code. The kernel `RandomGPU<<<32, 128>>>(d_Rand, n_per_rng)` gets called in the beginning of each iteration. That causes `d_Rand` array to be filled with fresh random numbers. It is very important to change the seed all the time by calling `seedMTGPU(rand()%2048)`, otherwise random numbers will be the same on every iteration.

Using Mersenne-Twister has a number of advantages and disadvantages. Because the code operates on the GPU, there is no memory overhead in copying random numbers. However, the fact that the generated numbers reside in the device memory should impose further penalties in accessing that memory. In reality though, memory latencies are hidden by massively parallel pipelines with hardware schedulers. There is also a scalability concern in storing all numbers as the device memory is limited. It is possible to solve all those problems by combining the `RandomGPU` kernel with the main `colonise` kernel to generate numbers on the fly. This approach however is not trivial and left as a future expansion.

3.1.4 Device code

Both kernels `colonise` and `update_pheromones` are located in `tsp_antcolony_kernel.cu` file. The `update_pheromones` kernel is trivial - simply checking if current thread (which symbolises an edge) is on the best path so far and applying Formula 2.2.

The `colonise` kernel has to first figure out the starting vertex(just pick up a random number) and initialise an array `visited` with `-1`'s (expect the starting vertex). Then for `n-1` number of times the ant has to choose the next vertex depending on the current vertex.

At this point, it turned out that it is best to use the state-transition rule (Formula 2.1) in a limited way in order to allow main paths to gain some more pheromone. Hence a major heuristic was introduced at this stage that improved the convergence rate severely. This is achieved by randomly choosing whether to **explore** paths using state-transition rule, or to **exploit** the edge with the most pheromone. This idea was suggested by the later chapters of Dorigo's book that were not read during the preparation stage of this study[5].

So, at each step each ant (thread) has to first randomly decide (with probability derived in the Evaluation chapter) whether to explore or exploit paths. If exploitation is chosen, then the ant would simply choose the outgoing edge with the most pheromone on it. If exploration option is chosen, then the ant does the following:

- take a random number which would be in the range `(0,1)`

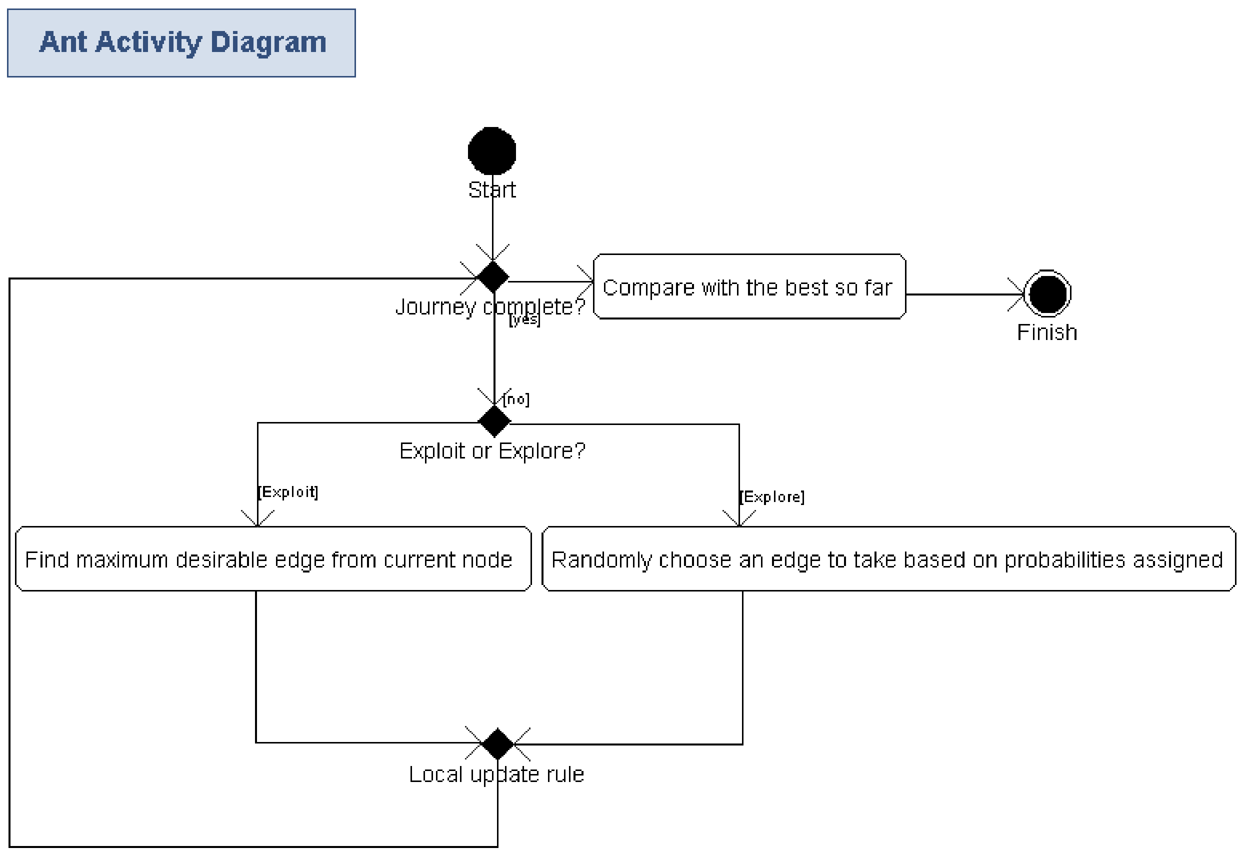
Ant Activity Diagram
Start
Journey complete?
[yes]
Compare with the best so far
Finish
[no]
Exploit or Explore?
[Exploit]
[Explore]
Find maximum desirable edge from current node
Randomly choose an edge to take based on probabilities assigned
Local update rule

Figure 3.2: Activity Diagram of an ant's actions inside of the `colonise` kernel.

- using Formula 2.1 iteratively sum up probabilities for each valid edge until the sum is larger than the random number taken
- the final edge is the next transition to take in accordance with probabilistic choice

After choosing the next edge to take, its weight is added to aggregator variable `sum`. Also local update rule (Formula 2.2) has to be applied to the corresponding edge. The procedure gets repeated again until `n-1` edges are collected and all vertices are visited. The final step to make would be to pick the edge connecting the starting vertex and the final vertex in order to form a cycle.

The best so far (`d_best`, stored in device memory for all threads to share) is then compared to the aggregate `sum` of the whole trip and updated if needed.

One minor problem is caused by the fact that CUDA does not allow dynamic memory allocation inside of a kernel. This forces one to create static-size arrays inside of a program. The `visited` array which stores the current path must be of size `n`. Because `n` is a parameter, it is only possible to create a static-size `visited` array. It was decided that a size large enough would be chosen to accommodate for all problem instances that would be met in the course of this study. This problem can of course be solved by preallocating `visited` arrays for every ant in the device memory, but this was left as a future extension as this fact does not affect the process of obtaining results.

3.2 CPU implementation

Writing the Java version of the same algorithm was trivial after writing the CUDA program. Some bits were simply copied because Java and C are so similar in their syntax. The overall UML-style diagram is pictured on Figure 3.3.

In Java implementation threads were again used to represent ants. In the CUDA version, Mersenne-Twister and global update rule were done on the GPU because of the memory transfer overhead. Whereas for the complete CPU implementation no memory transfer is needed, therefore those routines were not parallelised. Moreover, a simple Java `Random` class was used for generating random numbers because of its easy availability and obvious absence of concurrency problems.

One interesting implementation trick that was used to signal that all ants finished their trips is *Observer* design pattern explained on Figure 3.4. The key is to keep a counter of the number of ants that terminated. Once that counter reaches the total number of ants in the system, the main thread is notified to wake up. Notice that the main thread is put to wait on the *conditional variable* `(ANTSSOFAR==ANTS)` in general formal concurrency terms.

Responsibilities by class:

- **TSBLIBTest**: This class is the main entry to the program. It parses the input file in TSBLIB format and launches the colony, recording the time it takes to complete. This class was separated from other functionality to allow parsing of different formats that could be used with the program in future.
- **AntColony**: Contains the main `colonise` routine which simply creates a number of threads(ants) and handles `update` function of the Observer design pattern.
- **Ant**: This is a `Runnable` which in its run method performs actions of an ant for the whole trip. It is also responsible for updating the Observer(`AntColony`) on completion.
- **Graph**: All threads communicate with the main `Graph` object which simply stores all the data related to the problem instance: δ and τ matrices, τ_0 and δ_{best} values.

Java implementation was also used with a bash script similar to `runtsp.sh` from the GPU version to facilitate file management when launching multiple tests of the program.

Notice that all main Java programming paradigms like *loose coupling*, *tight encapsulation* and *high cohesion* are obeyed in the design of the component. This is to ensure that the system is extendable and usable in different environments.

During early test stages of the implementation it turned out that Java threads are heavyweight and it is not possible to create too many of them due to memory constraints. Java can therefore handle only hundreds of threads, whereas GPUs are suited to executing thousands of threads. This fact must be taken into consideration during the evaluation of results.

3.3 Extensions

None of the proposed extensions were implemented for this study due to lack of time and interest in them.

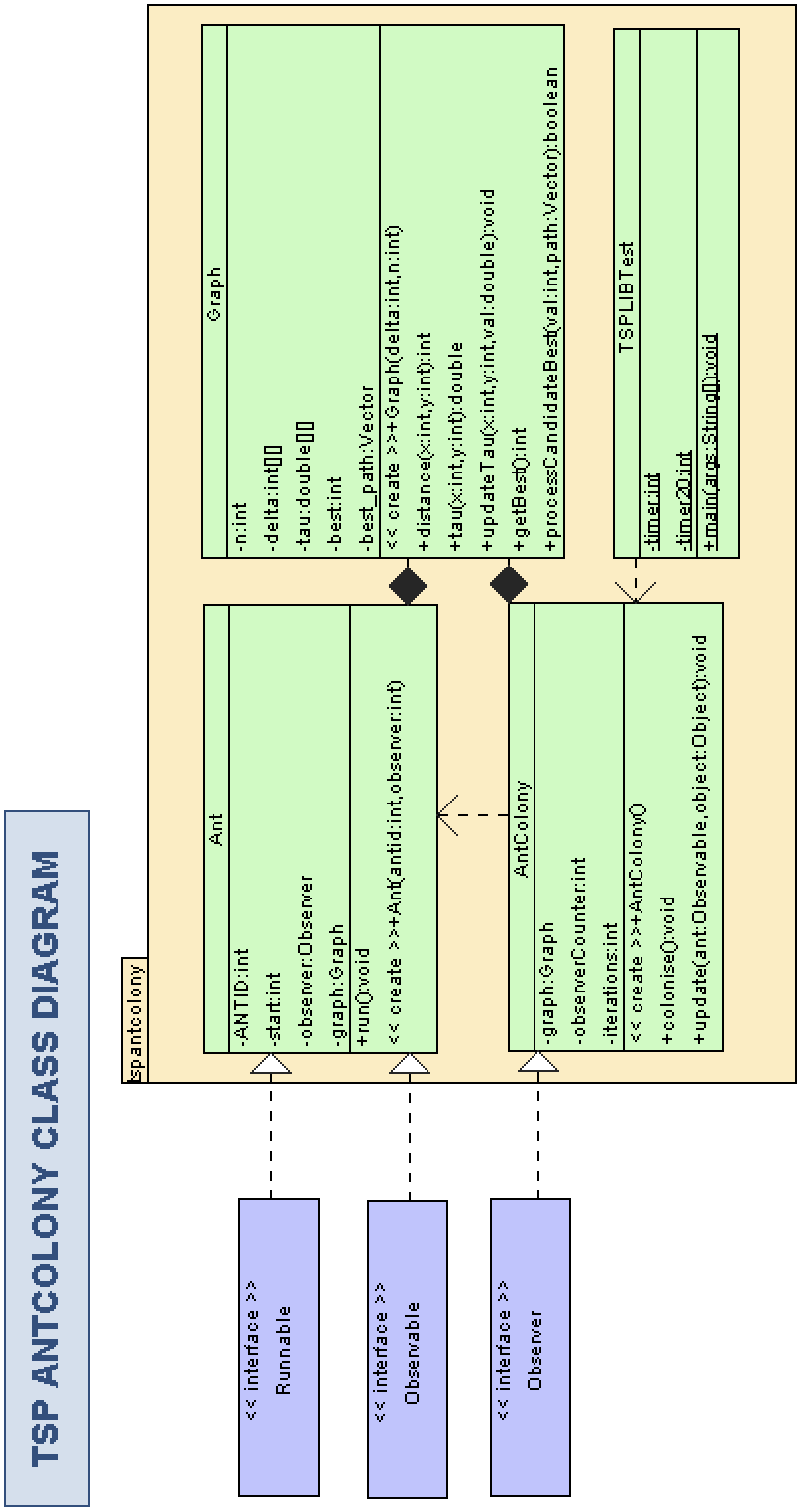

Figure 3.3: Class diagram of the Java version

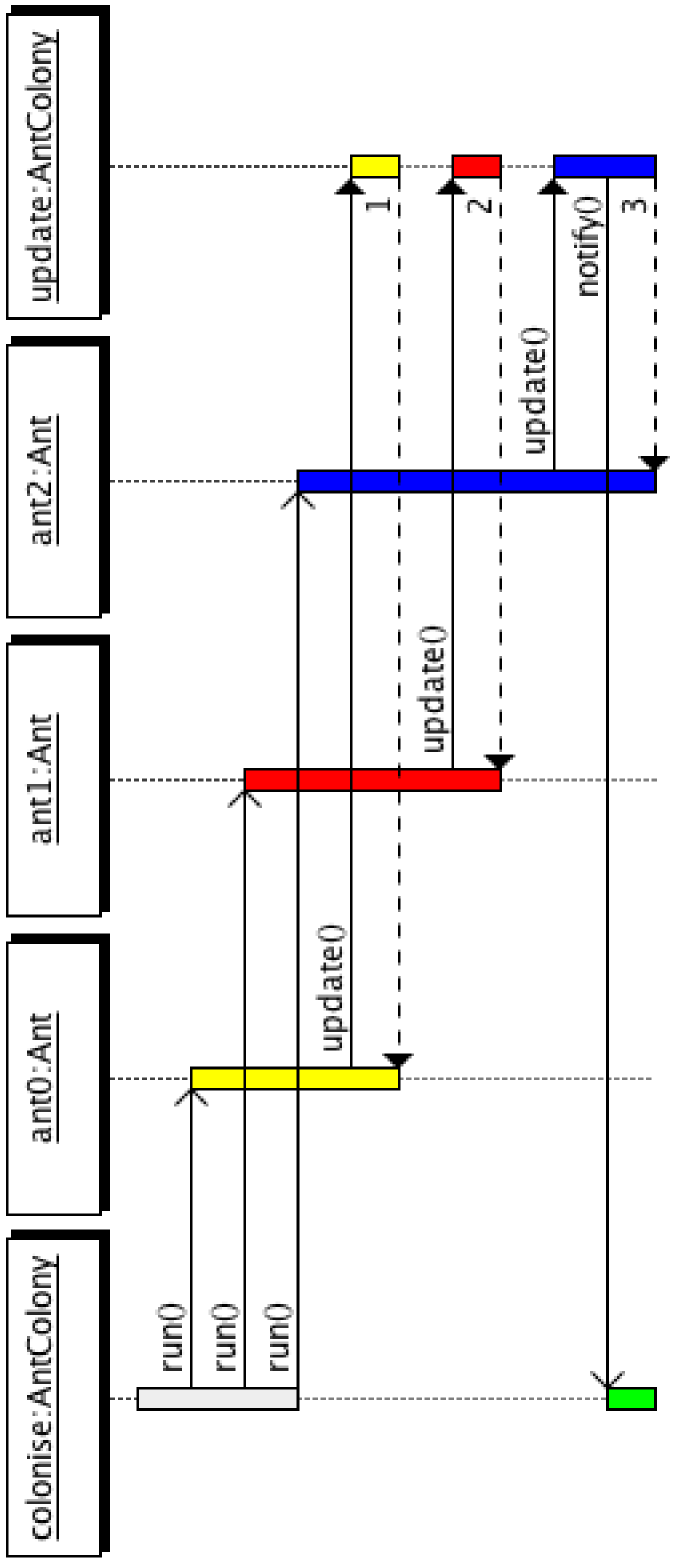

Figure 3.4: Sequence diagram for only 3 ants (for simplicity) showing the use of the Observer pattern

Chapter 4

Evaluation

4.1 Criticism of implementation

There are many points at which this implementation could be improved to show better results. In this sense a lot of measurements taken in this study are only valid for this implementation, not for the algorithm in general or the hardware involved. However, this study can still be a valuable resource for obtaining general trends and learning problems that occur with the current approach taken.

The GPU implementation was implemented properly and no bugs were observed. However, there are some issues to look at:

1. **Threads are ants**:
   Obviously, the approach taken may not reflect the most efficient form of performing computation. Different implementations on CUDA differentiate by how they approach parallel decomposition. Other implementation approaches could be taken, such as using parallel matrix manipulations (i.e. thread is an edge which is either taken or not by each ant). These approaches were not studied due to limited resources for this project, but there is no reason to regard them as less appropriate.

2. **No use of shared memory**:
   Most CUDA programs make use of shared memory, which is faster than device memory. Because of the way the ants were used, there was no obvious opportunity to use the shared memory to gain an additional speedup. Usually, shared memory is pre-filled with some data that is commonly used between threads in one block (a bit like a manually managed cache). In the case for this implementation, there was no special criterion by which ants were grouped into blocks, so it was not possible to predict what data would be requested by all ants in one block. Even if it were possible to divide the thread-space, there would be cohesion issues with making sure that ants use up-to-date information. The local update rule requires that updates are made instantaneously, because the whole point of the local update rule is to allow

little communication between ants during one iteration rather than in between as in the case with the global update rule. In a better implementation shared memory should be used in such a way that cohesion problems do not occur.

3. **Race condition**:
   There is one point in the GPU implementation that is vulnerable to a potential race condition:

   ```
   // compare the path with the best achieved so far
   if (cost < *d_best) {
   d_best = cost;
   for (int i=0; i<n; ++i) d_path[i] = visited[i];
   }
   ```

   The danger comes from the fact that the array `d_path` is not protected by any locking mechanism when its values are updated. If it happens that some other thread obtains another result that is better than out-of-date value of `d_best` then there is a chance that the `d_path` array is filled with incorrect values.

   This problem comes from CUDA's limitations in support of locking. It was hard to come up with some other scheme that achieves desired results without locking, so this was left out. One solution would be to move results aggregation into CPU host code to be done serially, but that would be reflected in performance as well. In fact this race condition is very unlikely to happen and even if it happens, it will only bring in incorrect actions by the global update rule, which can be easily fixed by the colony on next iterations of the algorithm.

The CPU implementation also has its own weaknesses:

1. **Language chosen**:
   Java is good and easy to use, but too slow for the purposes of measuring performance of CPU version of the program. Thread switching overhead might also be a very important issue to look at as Java is not capable of dealing with too many threads. A better approach would be to use C/C++ with *pthreads*, the only concern here being whether it is cross-platform enough as experiments are likely to be conducted on different machines.

2. **Process scheduler overhead**:
   The application was run in a usual operating system environment with many applications open at the same time. This brings in some uncertainty about whether the results obtained are legitimate as it is not known how much time the process has actually spent on the CPU performing useful computation. In a more rigorous attempt a dedicated server should be used which is isolated from other programs.

Perhaps a more general point that can be addressed to both CPU and GPU implementations is user-friendliness. The interface for using the program is unintuitive and not easy to comprehend. It is entirely command-line based. Moreover some parameters still need to modified in the source-code and recompiled to obtain the desired functioning. This was justified in the aims of the project by the fact that the study was not designed for serious usage, but more as a proof of concept for extracting performance information.

4.2 Experiment 1: Analysing operation

This section is about making sure that the program performs well and does what it is meant to do. One obvious thing to do is to run the program iteration-by-iteration and analyse what is achieved by each step. The GPU implementation was set up to run only one ant for this experiment. More ants will show better convergence times, which will make it harder to understand what happens in between. The graph size chosen is `5`, because it is easy to visualise in 2D and also easy to observe for a human eye. One downside is that the size is too small to observe any interesting dynamics of the algorithm. Figure 4.1 shows how pheromones change between iterations.
This figure can be explained as follows:

- Iteration 0 depicts the graph as it is in the very beginning with all edges carrying τ_0 amount of pheromone.
- Iteration 1 shows that the best answer was actually found on the very first iteration of this algorithm and the red edges are now bolder, having more pheromone on them.
- Iteration 5 shows that for the next four iterations the ant chose to explore other paths. No better answer was found, but other edges are now gaining pheromone even though they are not on the best path. The global update rule does its job here, not allowing the best path to gain too much pheromone, however it is still ahead of the others (which is desirable).
- Iteration 100 represents a converged state where all edges have gained their maximum pheromone. The system then stays roughly the same onwards.

Figure 4.2 is similar to the previous figure. In fact it shows a different run of the program, demonstrating that different patterns occur on every run. This time blue edges represent the best on the corresponding iteration, upon which the global update rule actually operates in this implementation. Notice that the algorithm is actually choosing between two routes that are close in their cost. The common edges of these paths gain more pheromone representing that those edges might be very important to include in a solution.

Overall, the operation of the implementation is sound in accordance with the theory for this project. Both CPU and GPU implementations show similar characteristics, but there are some differences that are discussed in Section 4.4 below. One characteristic of the

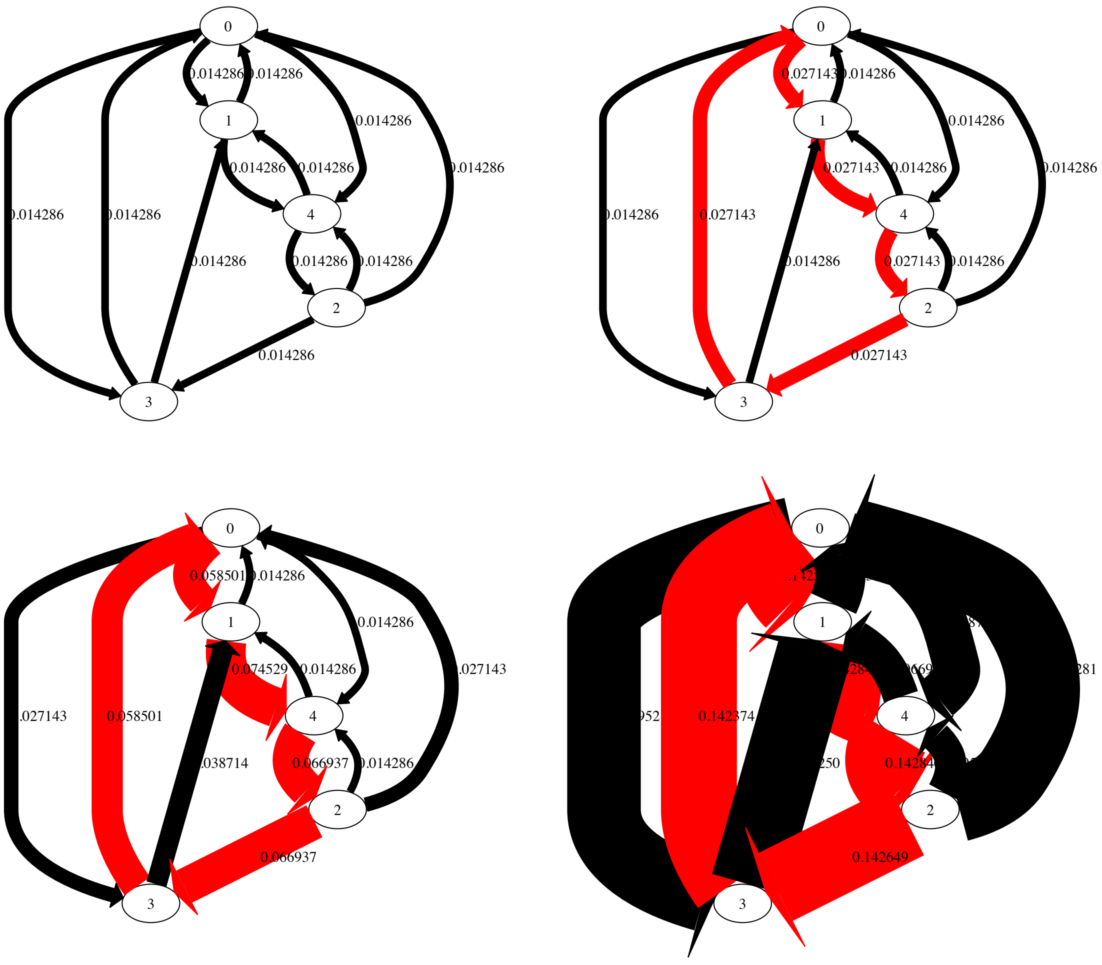

Figure 4.1: Pheromones represented as a graph after 0, 1, 5 and 100 iterations respectively. Red arrows represent the best path found globally. Performed with only one ant.

algorithm behaviour that could be desirable is that the actual best TSP cycle gets visibly noticeable as on Figure 4.10 which was performed without the global update rule which lowers the significance of the best path, also it had constant evaporation on all edges so that only pheromone on the best path so far could survive. However, this approach doesn't scale very well because it relies heavily on the initial guess of the best path which could lead the colony to converge to an incorrect answer. Another approach that could be studied (but was put aside in this project) is to stop applying the global update rule after some number of iterations. This however requires determining what iteration should that be for each problem instance.

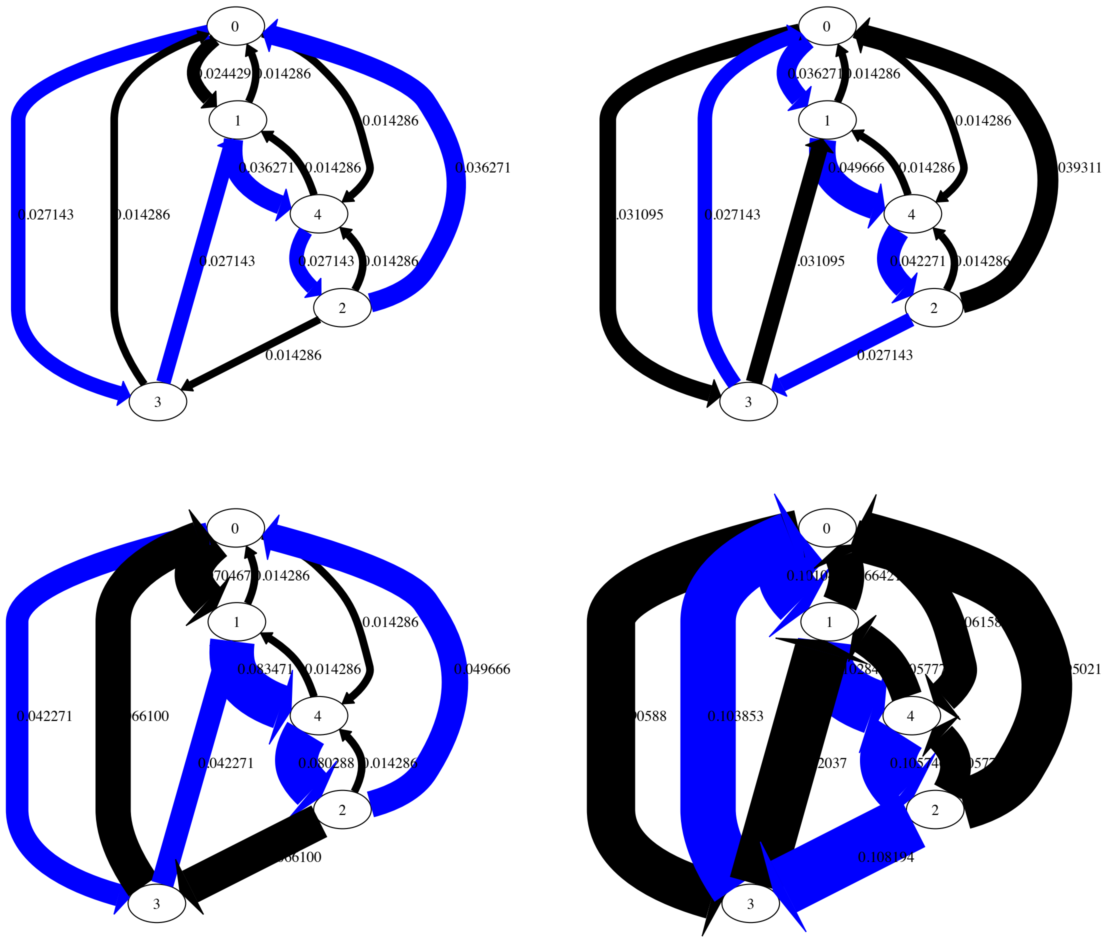

Figure 4.2: Pheromones represented as a graph after 2, 3, 7 and 100 iterations respectively. Blue arrows represent the best found on the last iteration. Performed with only one ant.

4.3 Experiment 2: Optimal number of ants

The second experiment aims to determine the optimal number of ants (that is threads in this implementation) to use in order to get answers efficiently. It is easy to observe that increasing the number of ants leads to increasing the importance of the local update rule as it gets more applications throughout the target graph.

Figure 4.3 shows how the time to reach target answer accuracy of a 100-node problem differs according to the number of ants. When analysing this graph one can notice that too few ants (roughly less than 100) result into bad performance. This could suggest that the number of ants should be at least equal to the number of vertices in the graph. The best performance is achieved between 100 and 1000 ants. The graph then shows a steadily decreasing performance when more than 1000 ants are involved.

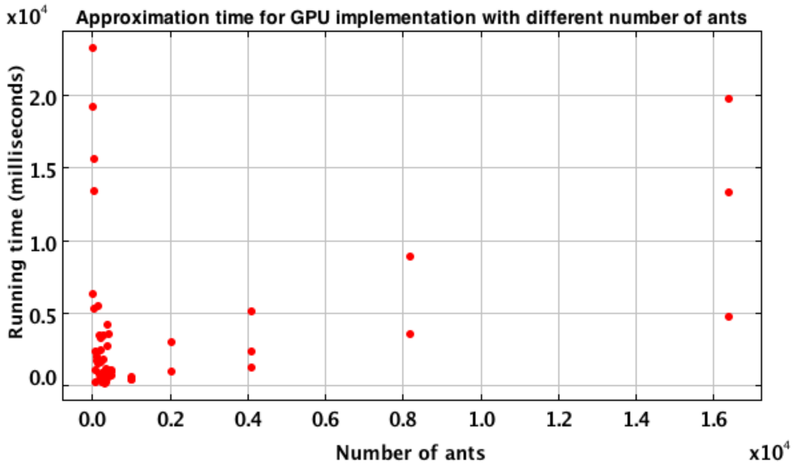

Figure 4.3: Graph showing the time taken to solve a 100-node problem (`TSPLIB.kro124p`) on GPU with different number of ants(three attempts each).

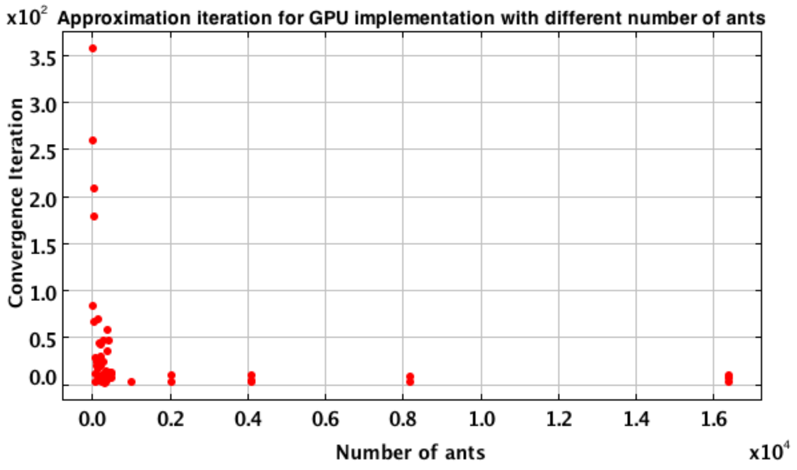

Figure 4.4: Graph showing the number of iterations to solve a 100-node problem (`TSPLIB.kro124p`) on GPU with different numbers of ants(three attempts each).

The decreasing performance with larger numbers of ants could be explained by Figure 4.4 which shows corresponding iteration counts at which the target approximation from Figure 4.3 was achieved. The figure clearly implies that a larger number of ants has a positive effect on lowering the convergence iteration count. However, one can notice that the iteration count does not change dramatically after about 1000 ants. This means that the performance decline is actually caused by the limitations of the algorithm itself in not converging any better when more ants are added. This results in diminishing returns when more and more ants are wasting their effort on something that would not improve anyway.

4.4 Experiment 3: Running time

The idea of this experiment is to observe how the program performs with different problem instances from the TSBLIB library.

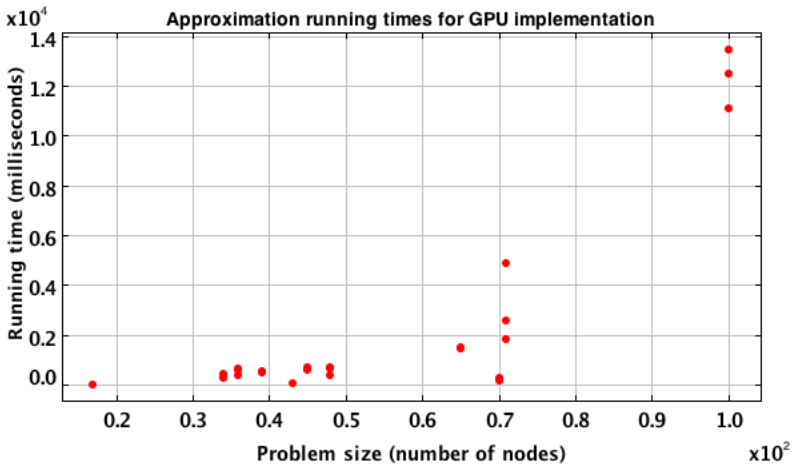

Figure 4.5: Graph showing times taken (in ms) to solve different size problems on GPU(three attempts per problem). Performed using 32 ants.

When it comes to analysing GPU performance, Figure 4.5 shows running time measurements for problem instances in the TSPLIB library of sizes up to 100. The first point that one can notice is that the graph is not smooth. Some problem instances are solved quicker than expected (such as 43, 70 and possibly 65). This could mean that problems in the TSPLIB library are not of uniform hardness. Some instances are easy to crack because of the heuristics implemented. Ignoring those cases, it is possible to observe a slowly rising trend which may be slightly shifted from the origin. This could be caused by a high probability of finding the optimal answer straight away, given the number of threads involved.

Once again, as in the previous section, it is important to look at the iteration count as well as the running time. Figure 4.6 shows the corresponding iteration counts for each problem instance. It is very interesting to see that the number of iterations actually stays pretty low at no more than about 350 iterations, growing steadily as the number of vertices increases. Notice that the running time has slightly different graph features, caused by the fact that the time complexity of ant's operation is $O(N^2)$, which makes up the overall time complexity to be $O(i(N) \times N^2)$, where $i(N)$ is an upper bound function on iterations. From the Figure 4.6 it is seems like the upper

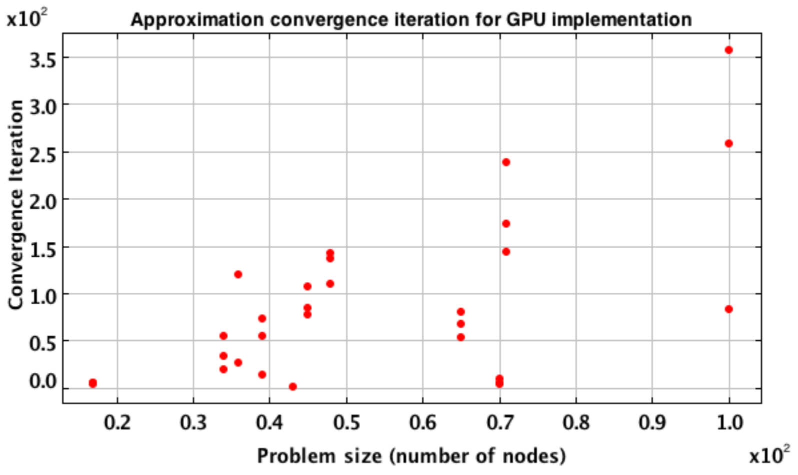

Figure 4.6: Graph showing the number of iterations of ACO to solve different size problems on GPU(three attempts per problem). Performed using 32 ants.

bound is linear, which would mean that the time complexity of the whole algorithm is polynomial. However, this conclusion cannot be derived in the context of that single plot as it is only an experimental result on a limited set of problem instances. Theoretical foundations for this claim have no grounds so far and are not discussed further in this study.

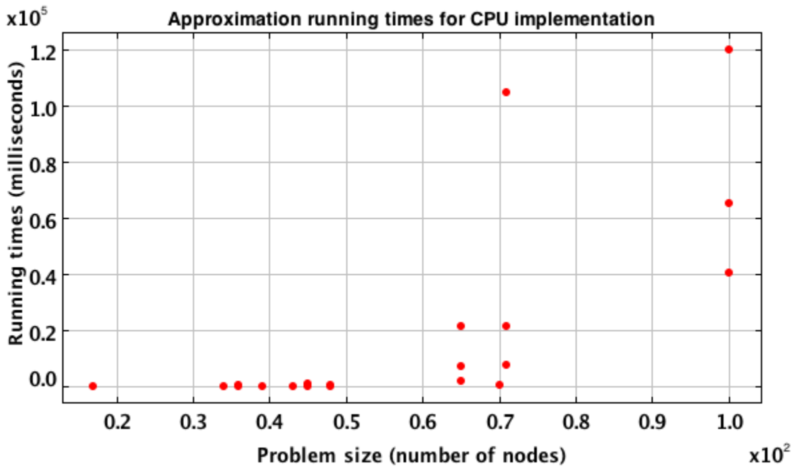

Figure 4.7: Graph showing times taken (in ms) to solve different size problems on CPU(three attempts per problem). Performed using 32 ants.

Figure 4.7 shows running time measurements for the corresponding CPU implementation of the algorithm in Java. What is strange about that program is that it performs extremely well when it operates on less than 50 nodes. This could be because of the heuristics used. The program performs extremely badly with more than 50 nodes. From the few data points that are available, it is possible to estimate that the speedup of using a GPU is around $\times 9$ when used with 32 ants. However, the GPU should be much better than CPU in general because it was shown above that the number of ants severely affects the convergence rate. The CPU version with its heavy threads can only handle about a hundred of threads, whereas the GPU version was tested to work well even with many thousands of threads. Hence, GPU has a potential to tackle a larger set of problem instances.

4.5 Scalability

In order to analyse scalability characteristics of the ACO algorithm additional experiments were conducted on another machine: NVIDIA GeForce 9600GT with 64 CUDA cores and having Intel i7 CPU with 8 cores @ 3.20GHz. This experiment is aimed at demonstrating the scale of improvement when tested on increasing number of cores. The reason why this experiment is really hard to perform is that one has to have many different cards that all have different number of CUDA cores. That is just too expensive to do, but in this case looking at just two cards that are sufficiently different may provide some basis for understanding overall scalability trends.

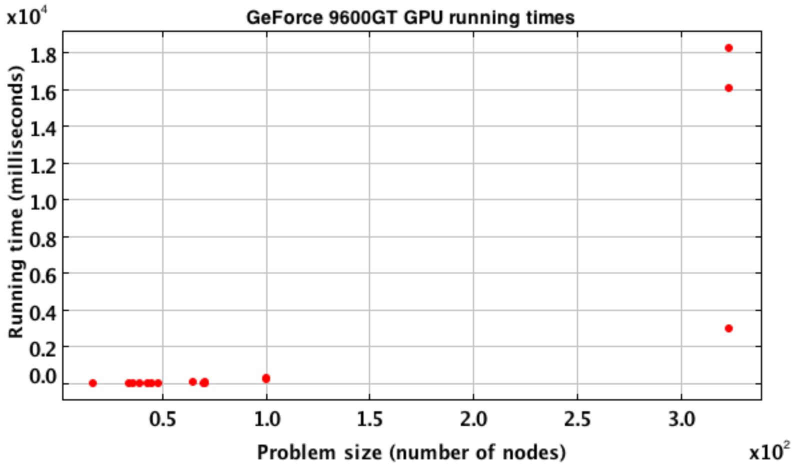

Figure 4.8: Time taken to solve problems on NVIDIA GeForce 9600GT with 512 ants

Figure 4.8 shows the running times that demonstrate an extremely good speedup compared to previous results. One can notice that the algorithm has now even managed to solve a 323-node problem instance in about 15 seconds, which is a significant achievement.

9400M solves a 100-node problem instance using 512 ants in just about 1000ms (see Figure 4.3), whereas 9600GT does the same in about 300ms. Therefore, $speedup = \frac{1000}{16} / \frac{300}{64} \simeq 13.3$ per core. At this point, it is possible to anticipate that the algorithm scales extremely well. It is however possible to suggest that 9400M is a mobile GPU and therefore may be fundamentally slower than desktop GPUs with same number of cores. Even though the current implementation is not heavily bottlenecked on the memory, it is also important to note that the memory bandwidth of 9400M is just 21GB/s, compared to 57.6 GB/s for 9600GT.

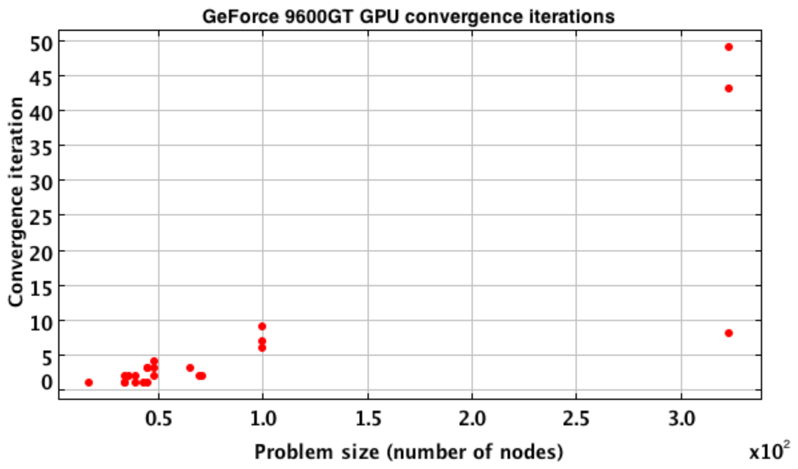

Figure 4.9: Number of iterations taken to solve problems on NVIDIA GeForce 9600GT with 512 ants

Figure 4.9 demonstrates that measurements taken are similar to rising trends set by the Figure 4.6. It takes about 10 iterations for the 9400M run to get to the target of the same 100-node problem (see Figure 4.4) and it takes almost the same (slightly less) for the 9600GT run. So, convergence iteration does not scale as much as running time does. This confirms further that reducing the convergence iteration is the key to obtaining better performance of ACO, regardless of the parallelism. The gain from adding more cores on a GPU comes from the ability of creating more ants, which in turn improves convergence time.

4.6 Other issues

It is interesting to see how CPU and GPU performances differ in their results. One interesting issue to think about is the models of parallel computation used on both computation devices. Java threads are usually software threads which can occasionally be supported by hardware that does multithreading. In general CPU parallel computation model is only abstracted away from the actual serial computation model by means of threads and schedulers. On the other hand, GPU computation model is explicitly parallel where it is known that a certain number of threads (Number of CUDA cores $\times$ Warp size) will be executed at the same time in SIMD style, so the computation is not exactly serial. The implication of this difference in models is mainly reflected in the use of the local update rule. It is guaranteed in the GPU implementation that a certain number of threads will apply the local update rule at the same time (using gather and scatter capabilities of the hardware to simultaneously access memory), thus not taking into account each other's local updates. The same happen in the Java implementation, but this is rather unlikely because the CPU is not SIMD. Hence the difference in operation of ACO on two models.

None of the solutions (GPU or CPU) could solve any larger problem instance than the 323-node `TSPLIB.rbg323`. In fact only 9600GT could tackle `rbg323`, 9400M could not. The next problem instance which is available in the TSPLIB library is `rbg358` which contains 358 nodes and does not converge in the time limit of 3 minutes on both 9400M and 9600GT. This fact is rather disappointing and leads to a conclusion that Ant Colony Optimisation may not have been the best algorithm to use for this project if performance of a single card was vital. Scalability results bring hope, but not convincing enough. On the other hand, different implementations of the same algorithm might lead to different results as it all depends on the quality of implementation. However, since the aim of the project is to investigate parallelisation of the TSP, and the ACO algorithm is very well-suited for parallelism and minimum locking is required.

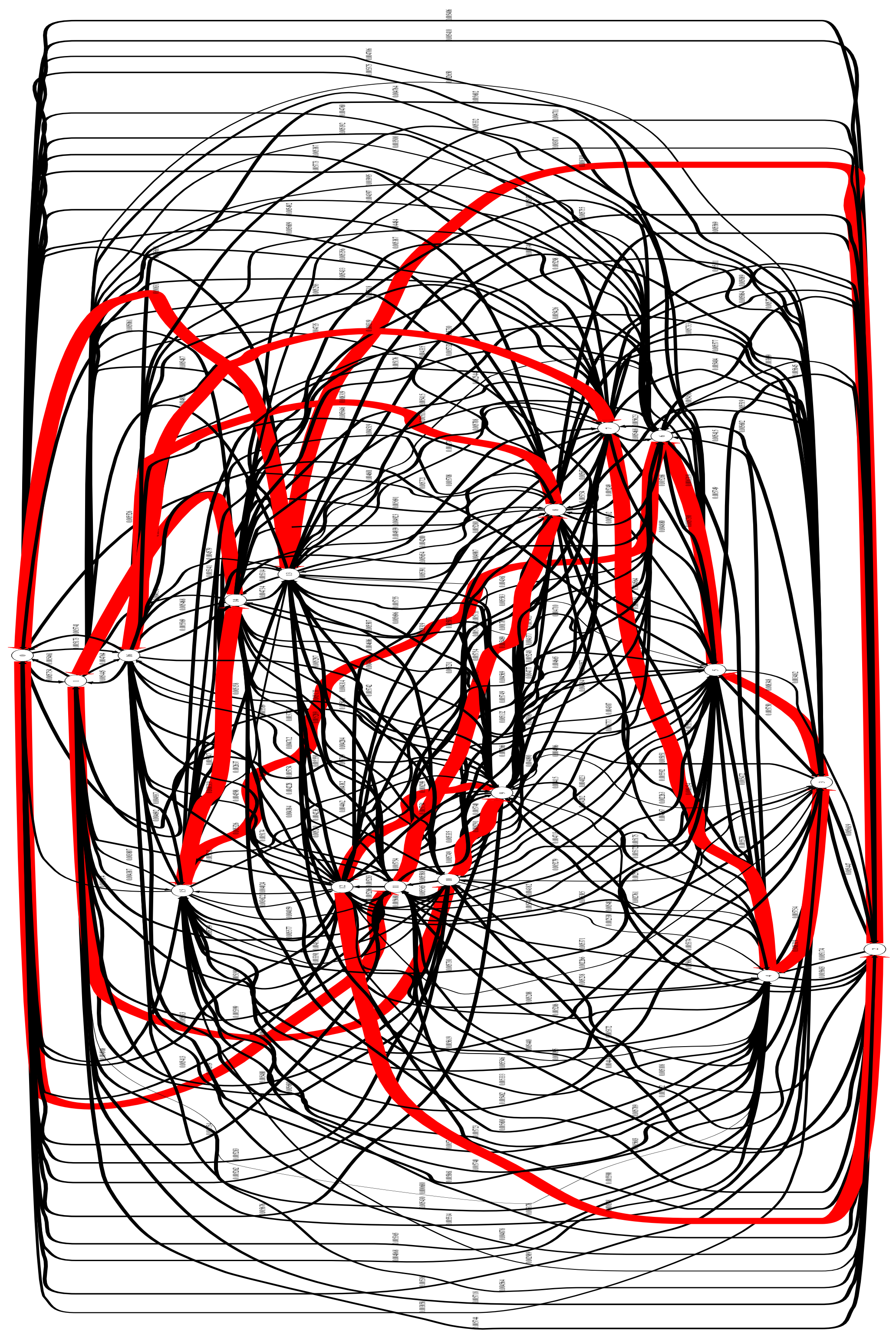

Figure 4.10: A visualisation of the pheromone matrix of a 17-node problem instance ((`TSPLIB.br17`)) solved in 2500 iterations. Red arrows represent the best cycle found. Performed without the global update rule, but with global evaporation of pheromones on all edges.

Chapter 5

Conclusion

Overall the project was a success. The main aim was to investigate GPU performance when applied to a problem like TSP. The extensive preparation taken before starting the implementation phase helped to organise thoughts together and understand possible directions for implementation. It was therefore decided that Ant Colony Optimisation algorithm is the most promising one from the parallelisation perspective. It was later justified in the evaluation phase that the algorithm indeed is able to solve some fairly challenging problem instances involving large datasets.

All three success criteria were met. The first one being just a coding criteria. Not only the implementation was carefully produced to reflect the purpose of this study, but also it was evaluated to find criticisms about it and possible ways of fixing them in future work. The second success criterion was about scalability. The algorithm has shown very promising scalability characteristics (13.3 per core) when tested on a more powerful GPU. Finally, it was vital to show that GPU offers a significant speedup over CPU. That was demonstrated clearly and the speedup gained was approximated to be around $\times 9$ for the instances that were studied.

The structure of the project that was proposed initially was kept the same and the whole project delivery was flowing according to the plan. The proposed extension was not implemented due to lack of interest in implementing another Map-Reduce approach because it turned out that the algorithm chosen depends more on heuristics and other features such as number of ants. Implementing another solution is likely to produce similar results, however it is still interesting to study how to use Map-Reduce on GPUs when applied to a complex problem like TSP, so this was left as a future extension to this study that can be developed further outside of this project.

During the course of the project there were some interesting results obtained that were not explained in the Dorigo's book [5]. For instance, the book said that it was not known whether parallelisation of the ACO algorithm would yield better results. $\times 9$ speedup and 13.3 scale factor gained with exactly the same number of ants could suggest that parallelisation is actually an improvement. The book also claimed that the optimal number of ants to use should be either 64 or equal to the number of vertices in the graph. It was shown quantitatively that increasing the number of ants improves convergence, while too many ants cause diminishing returns as an underlying reason.

The project aims were met, however there is also one potential aim that is desirable, but deliberately was not initially included because it could have been too hard to achieve. That aim is the actual real-world applicability of the project. In reality, it might be useful to use a program such as developed for this study in businesses of large logistics companies or any other companies that depend on optimisation problems heavily. A real-world problem may consist of thousands of nodes with all sorts of externalities that the program developed here would not be able to solve. It is known that there are other efficient implementations that perform much better than the approach developed in this study. Of course, those implementations are likely to use very complicated heuristics, where this project only uses a few. Another reason could be that the algorithm chosen (ACO) is suitable for TSP, but must be implemented differently. It is the case though that ACO would perform reasonably well on constantly changing graphs. A more disappointing issue is that running times of ACO are non-deterministic and rely heavily on the quality of random numbers produced. On the other hand, TSP is an NP-Complete problem, so it would be ambitious to expect too much from this approach and getting to a 20% accuracy is still very hard to guarantee on every single problem instance.

Bibliography

[1] Tsplib, 1995. Available at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[2] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. Concorde. 1999. Available at http://www.math.princeton.edu/tsp/concorde.html.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[4] M. Dorigo and T. Stutzle. A short convergence proof for a class of ant colony optimization algorithms. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 6, NO. 4*, 2002.

[5] M. Dorigo and T. Stutzle. *Ant Colony Optimization*. MIT Press, 2004.

[6] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[7] M. Hahsler and K. Hornik. Tsp infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 2007.

[8] P. Harish and Narayanan P.J. Accelerating large graph algorithms on the gpu using cuda. 2007.

[9] M. Held and R.M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 1962.

[10] J.V. Leeuwen. *Algorithms and Complexity*. Elsevier, 1990.

[11] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 1973.

[12] J.F. Pekny and D.L. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Proceedings of the 1990 ACM annual conference on Cooperation*, 1990.

[13] V. Podlozhnyuk. Parallel mersenne twister, 2007.

Appendices

Appendix A

GPU Source Code

Only two key files from the program are presented. The rest is trivial, including the whole Java implementation.

A.1 tsp_antcolony.cu

```
/*
 * Copyright 2010 Ruslan Kudubayev.
 */

/*
 * Host code.
 */

// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, project
#include <cutil_inline.h>

// includes, kernels
#include <MersenneTwister_kernel.cu>
#include <tsp_antcolony_kernel.cu>

#include <helpers.cu>

FILE *infile;

//performs a nearest neighbour simple greedy search to get a value for tau0.
float nearest_neighbour(int* h_A, int size) {
        int cur = 0;
        int visited[WA];
        float res = 0;
        for (int i=0; i<size; i++) visited[i] = 0;
        visited[cur] = 1;
        for (int i=1; i<size; i++) {
                int min = 214748364;
                int minj = i;
                for (int j=0; j<size; j++) if (visited[j] == 0) {
                        if (h_A[size*cur+j] < min) {
                                min = h_A[size*cur+j];
                                minj = j;
                        }
                }
                res += min;
                visited[minj] = 1;
                cur = minj;
        }
        res += h_A[size*cur + 0];
        return res;
}

//Align a to nearest higher multiple of b
extern "C" int iAlignUp(int a, int b){
    return ((a % b) != 0) ?  (a - a % b + b) : a;
}

//ceil(a / b)
extern "C" int iDivUp(int a, int b){
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}

extern "C" void initMTRef(const char *fname);

////////////////////////////////////////////////////////////////////////////////
// Run test
////////////////////////////////////////////////////////////////////////////////
void runTest(int argc, char** argv)
{
        char* filenametotest;
        int targetanswer;
        if (argc > 2) {
                // parameters to the executable:
                // 1. filename
                // 2. the answer to the problem as an integer
                filenametotest = argv[1];
                targetanswer = atoi(argv[2]);
        }
        // get the size of the problem from the file first
        const int n = getSize(filenametotest);

        // dividing the number of ants evenly into blocks for computation
        int blocks = iDivUp(ANTS, BLOCK_SIZE);
        printf("Blocks: %d\n", blocks);

        // allocate host memory for matrice A
        unsigned int size_A = n * n;
        unsigned int mem_size_A = sizeof(int) * size_A;
        int* h_A = (int*) malloc(mem_size_A);
        getGraph(h_A, n);

        cudaSetDevice( cutGetMaxGflopsDeviceId() );

        // set seed for rand()
        srand((unsigned)time(NULL));

        /*
        This routine is responsible for preparing to generating random numbers on the device
        and leaving them there for the other kernel to use.
```

```
*/
float *d_Rand;

int path_n = (2*n-1) * BLOCK_SIZE * blocks;

int n_per_rng = iAlignUp(iDivUp(path_n, MT_RNG_COUNT), 2);
int rand_n = MT_RNG_COUNT * n_per_rng;

//printf("Initializing data on the device for %i random samples...\n", path_n);
cutilSafeCall( cudaMalloc((void **)&d_Rand, rand_n * sizeof(float)) );
initMTRef("MersenneTwister.raw");
loadMTGPU("MersenneTwister.dat");


// allocate device memory for storing delta matrix
int* d_A;
cutilSafeCall(cudaMalloc((void**) &d_A, mem_size_A));
// copy host memory to device
cutilSafeCall(cudaMemcpy(d_A, h_A, mem_size_A,
                        cudaMemcpyHostToDevice) );

// allocate device memory for tau matrix
unsigned int size_C = n * n;
unsigned int mem_size_C = sizeof(float) * size_C;
float* d_C;
cutilSafeCall(cudaMalloc((void**) &d_C, mem_size_C));
// allocate host memory for the tau on host
float* h_C = (float*) malloc(mem_size_C);
//initialise pheromones tau matrix
float tau0 = 1.0f/((float)n * nearest_neighbour(h_A, n));
//printf("tau0: %1.20f\n", tau0);
for (int i=0; i<size_C; ++i) h_C[i] = tau0;
// copy host memory to device
cutilSafeCall(cudaMemcpy(d_C, h_C, mem_size_C,
                        cudaMemcpyHostToDevice) );

// allocate device memory for the path vector
int size_P = n;
int mem_size_P = sizeof(int) * size_P;
int* d_P;
cutilSafeCall(cudaMalloc((void**) &d_P, mem_size_P));
// allocate host memory for the R on host
int* h_P = (int*) malloc(mem_size_P);
// copy host memory to device
cutilSafeCall(cudaMemcpy(d_P, h_P, mem_size_P,
                        cudaMemcpyHostToDevice) );

// allocate device memory for best on the iteration
int* d_best;
cutilSafeCall(cudaMalloc((void**) &d_best, sizeof(int)));
// allocate host memory for the best on host
int* h_best = (int*)malloc(sizeof(int));
*h_best = 2147483647;
int global_best = 2147483647;
// copy host memory to device
cutilSafeCall(cudaMemcpy(d_best, h_best, sizeof(int),
                        cudaMemcpyHostToDevice) );

// create and start timer
unsigned int timer = 0;
cutilCheckError(cutCreateTimer(&timer));
```

```
    cutilCheckError(cutStartTimer(timer));

    // dimensions of the global update kernel
    dim3 threads2(BLOCK_SIDE_UPDATER, BLOCK_SIDE_UPDATER);
    int side_blocks = n/BLOCK_SIDE_UPDATER;
    if (n%BLOCK_SIDE_UPDATER != 0) side_blocks++;
    dim3 grid2(side_blocks, side_blocks);

    // **************************************************
    // the main block of code that executes the kernel.
    int firsttimeto20 = 0;
    for (int iteration=0; iteration<2048; ++iteration) {
        //generate random numbers for this iteration
        seedMTGPU(rand()%100000);
        RandomGPU<<<32, 128>>>(d_Rand, n_per_rng);
        cutilCheckMsg("RandomGPU() execution failed\n");
        cutilSafeCall( cudaThreadSynchronize() );

        // this is actually the rho and phi taken to be the same.
        float damping = 0.1f;

        // execute the kernel
        colonise<<< blocks, BLOCK_SIZE >>>(d_C, d_A, d_Rand, d_best,
                               d_P, n, damping, tau0);
        cutilSafeCall( cudaThreadSynchronize() );

        // get the best so far
        cutilSafeCall(cudaMemcpy(h_best, d_best, sizeof(int),
                     cudaMemcpyDeviceToHost) );
        cutilSafeCall(cudaMemcpy(h_P, d_P, mem_size_P,
                     cudaMemcpyDeviceToHost) );
        cutilSafeCall(cudaMemcpy(h_C, d_C, mem_size_C,
                     cudaMemcpyDeviceToHost) );

        // record when we get to 20% accuracy
        if ((firsttimeto20 == 0) && (((float)*h_best/targetanswer) <= 1.2f)) {
            firsttimeto20 = 1;
            printf("First time to 20 percent accuracy: %f (ms) \n",
                        cutGetTimerValue(timer));
            printf("Convergenece interation: %d\n", iteration+1);
        }
        // if reached the optimal answer then quit, no point to work anymore.
        if (*h_best == targetanswer) break;

        // global updating rule here.
        // can just execute another kernel here which would do that.
        // the reason is to not copy the data but do modifications over there.
        update_pheromones<<< grid2, threads2 >>>(d_C, d_best, d_P, n, damping);
        cutilSafeCall( cudaThreadSynchronize() );

        if (*h_best < global_best) {
            global_best = *h_best;
        }
        *h_best = 2147483647;
        cutilSafeCall(cudaMemcpy(d_best, h_best, sizeof(int),
                     cudaMemcpyHostToDevice) );
    }
    // **************************************************

    // stop and destroy timer
    cutilCheckError(cutStopTimer(timer));
```

```
        printf("Processing time: %f (ms) \n", cutGetTimerValue(timer));

        cutilCheckError(cutDeleteTimer(timer));

        // check if kernel execution generated and error
        cutilCheckMsg("Kernel execution failed");

        // copy result from device to host
        cutilSafeCall(cudaMemcpy(h_C, d_C, mem_size_C,
                              cudaMemcpyDeviceToHost) );
        cutilSafeCall(cudaMemcpy(h_P, d_P, mem_size_P,
                              cudaMemcpyDeviceToHost) );
        cutilSafeCall(cudaMemcpy(h_best, d_best, sizeof(int),
                              cudaMemcpyDeviceToHost) );

        printf("Result: %d\n", global_best);

        //printf("Tau:\n");
        //printArr(h_C,n);

        //outputfordotformati("original_graph.dot",h_A,h_P,n,0.1f);
        outputfordotformatf("tau_graph.dot",h_A,h_C,h_P,n,1200.0f);

        // clean up memory
        free(h_A);
        free(h_C);
        free(h_P);
        free(h_best);
        cutilSafeCall(cudaFree(d_A));
        cutilSafeCall(cudaFree(d_C));
        cutilSafeCall(cudaFree(d_Rand));
        cutilSafeCall(cudaFree(d_P));
        cutilSafeCall(cudaFree(d_best));

        cudaThreadExit();
}

////////////////////////////////////////////////////////////////////////////////
// Program main
////////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
        runTest(argc, argv);
        cutilExit(argc, argv);
}
```

A.2 tsp_antcolony_kernel.cu

```
/*
 * Copyright 2010 Ruslan Kudubayev.
 */

/*
 * Device code.
 */

#ifndef _TSP_ANTCOLONY_KERNEL_H_
#define _TSP_ANTCOLONY_KERNEL_H_

#include <stdio.h>
```

```
#include "tsp_antcolony.h"

////////////////////////////////////////////////////////////////////////////////
// colonisation
////////////////////////////////////////////////////////////////////////////////
__global__ void colonise(float* C, int* A, float* Rand, int* d_best, int* d_path, const int n,
                         const float R, const float tau0) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int antid = ((bx+by)*BLOCK_SIZE+(tx+ty));

    //calculate the offset in the Rand array
    int offset = antid*(2*n-1);

    float ra = Rand[offset + 2*n-2];

    int startNode = ((int)(ra*n))%n;
    int curNode =  startNode;
    int visited[WA];
    for (int i=0; i<n; ++i) visited[i] = -1;
    visited[startNode] = 0;
    int cost = 0;
    int collected;

    // in a while loop do movements according to the matrix updating tau
    for (collected=1; collected<n; ++collected) {
        int nNode = -1;
        // get random
        ra = Rand[offset + collected*2 - 2];
        //exploit or explore
        if (ra > 0.2f) { //exploit the paths, get the max one simply
                float max = -1.0f;
                int first = 1;
                for (int i=0; i<n; ++i) if (visited[i]==-1 && i!=curNode) {
                        float eeta, phe;
                        if (A[curNode*n+i]==0) eeta = 1.1f;
                        else eeta = (1.0f/A[curNode*n+i]);
                        phe = C[curNode*n+i] * eeta;
                        if (first || phe > max) {
                                max = phe;
                                nNode = i;
                                first = 0;
                        }
                }
                } else { // explore properly
                float sum = 0.0f;
                float sump = 0.0f;
                // calculate the sum
                for (int i=0; i<n; ++i) if (visited[i] == -1 && i!=curNode) {
                        // take care of the zero divisions...
                        float eeta;
                        if (A[curNode*n+i]==0) eeta = 1.1f;
                        else eeta = (1.0f/A[curNode*n+i]);
                        sum += C[curNode*n+i] * eeta;
                }
                //generate a random number
                ra = Rand[offset + collected*2 - 1];
                float target = ra * sum; // precalculate this for the p formula division
                // calculate the probability and jump if that probability occurs this time.
                for (int i=0; i<n; ++i) if (visited[i] == -1 && i!=curNode) {
                        // calculate the probability as per the equation before.
                        float p;
                        float eeta;
                        if (A[curNode*n+i]==0) eeta = 1.1f;
                        else eeta = (1.0f/A[curNode*n+i]);
                        // p calculated here with squaring eeta for better results
                        p = (C[curNode*n+i] * eeta);
                        if (target>sump && target<=p+sump) {
                                // yes. move.
                                nNode = i;
                                break;
                        }
                        nNode = i;
                        sump += p;
                }
        }
        if (nNode >= 0) {
                // accept the next node
                cost = cost + A[curNode*n+nNode];
                visited[curNode] = nNode;
                // apply local updating rule right now.
                C[curNode*n+nNode] = (1.0f - R) * C[curNode*n+nNode] + tau0;
                // move on
                curNode = nNode;
        } else {
                // don't really prefer to go there
                // this means that an ant has arrived
                // into some deadlock where it is best to die than
                // lead anyone else here.
                break;
        }

    }

    if (collected == n) {
        cost = cost + A[curNode*n+startNode];
        visited[curNode] = startNode;
        C[curNode*n+startNode] = (1.0f - R) * C[curNode*n+startNode] + tau0;
        // after done, compare the path with the best achieved so far
        if (cost < *d_best) {
                *d_best = cost;
                for (int i=0; i<n; ++i) d_path[i] = visited[i];
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
// global update
////////////////////////////////////////////////////////////////////////////////
__global__ void update_pheromones(float* C, int* d_best, int* d_path, const int n, const float A) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int i = (bx * BLOCK_SIDE_UPDATER) + tx;
    int j = (by * BLOCK_SIDE_UPDATER) + ty;

    if (i<n && j<n) {
        // update pheromones.
        float deposition = 0.0f;
        float evaporation = C[i*n+j] * (1.0f - A);
        if (d_path[i] == j) {
                        deposition = A/(*d_best);
        }
        C[i*n+j] = evaporation+deposition;
    }
}
#endif // #ifndef _TSP_ANTCOLONY_KERNEL_H_
```

Appendix B

Data collected

The data is presented in the form of Python source files that were fed into jHepWork application in order to produce plots.

B.1 GPU Running Times with 32 ants

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *

c1 = SPlot("Approximation running times for GPU implementation",600,400)
c1.visible()
c1.setNameY("Running time (milliseconds)")
c1.setNameX("Problem size (number of nodes)")
c1.draw( "", (17,17,17,34,34,34,36,36,36,39,39,39,
        43,43,43,45,45,45,48,48,48,65,65,65,
        70,70,70,71,71,71,100,100,100,
        ),
        (9.66,11.75,9.78,448.07,277.74,313.06,597.05,355.91,668.52,
        526.83,521.58,513.12,52.73,51.73,52.73,645.49,694.32,583.35,
        679.33,408.58,646.69,1518.10,1486.58,1471.52,255.57,268.63,193.81,
        2600.16,1813.13,4902.84,11108.29,13449.70,12504.22) )
```

B.2 GPU Convergence Iterations with 32 ants

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *

c1 = SPlot("Approximation convergence iteration for GPU implementation",600,400)
c1.visible()
c1.setNameY("Convergence Iteration")
c1.setNameX("Problem size (number of nodes)")
c1.draw( "", (17,17,17,34,34,34,36,36,36,39,39,39,
        43,43,43,45,45,45,48,48,48,65,65,65,
        70,70,70,71,71,71,100,100,100
        ),
        (5,3,5,19,55,33,26,27,119,54,14,73,
        1,1,1,84,107,77,110,137,142,53,80,67,
         4,5,10,143,174,238,258,83,357) )
```

B.3 GPU Running Times with Different Number of Ants

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *

c1 = SPlot("Approximation time for GPU implementation with different number of ants",600,400)
c1.visible()
c1.setNameY("Running time (milliseconds)")
c1.setNameX("Number of ants")
c1.draw( "", (32,32,32,64,64,64,96,96,96,
         128,128,128,160,160,160,192,192,192,
         224,224,224,256,256,256,288,288,288,
         320,320,320,352,352,352,384,384,384,
         416,416,416,512,512,512,1024,1024,1024,
         2048,2048,2048,4096,4096,4096,8192,8192,8192,
         16384,16384,16384
         ),
         (6312,23232,19151,5235,15571,13410,1011,232,2321,
         1711,1951,2197,5452,1771,1464,3440,755,426,
         3253,2383,1559,223,815,660,1813,218,3428,
         215,319,116,222,319,1161,484,2660,4141,
         3506,661,1056,1034,964,656,384,610,382,
         2924,945,945,1242,2358,5116,8866,3518,3508,
         4737,13280,19741) )
```

B.4 GPU Convergence Iterations with Different Number of Ants

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *

c1 = SPlot("Approximation iteration for GPU implementation with different number of ants",600,400)
c1.visible()
c1.setNameY("Convergence Iteration")
c1.setNameX("Number of ants")
c1.draw( "", (32,32,32,64,64,64,96,96,96,128,128,128,
         160,160,160,192,192,192,224,224,224,256,256,256,
         288,288,288,320,320,320,352,352,352,384,384,384,
         416,416,416,512,512,512,1024,1024,1024,
         2048,2048,2048,4096,4096,4096,8192,8192,8192,
         16384,16384,16384
         ),
         (259,357,83,178,208,66,28,2,11,26,23,20,
         17,21,69,4,8,44,19,30,42,7,9,2,
         2,23,46,1,3,2,14,3,2, 5,35,57,
         12,7,47,7,11,12,2,2,2,
         10,3,3,2,4,9,8,3,3,2,6,9 ) )
```

B.5 CPU Running Times with 32 ants

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *

c1 = SPlot("Approximation running times for CPU implementation",600,400)
c1.visible()
c1.setNameY("Running times (milliseconds)")
c1.setNameX("Problem size (number of nodes)")
c1.draw( "", (17,17,17,34,34,34,345,45,45,
         48,48,48,65,65,65,70,70,70,
         71,71,71,100,100,100
         ),
         (19,31,16,181,210,95,218,106,136,
         65,55,166,64,69,65,892,330,115,
         443,207,165,21296,7278,1832,
         220,218,335,7459,21211,104880,120116,65349,40301) )
```

B.6 GPU Running Times on a 64-core card

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *
from jhplot.math.StatisticSample import randUniform

c1 = SPlot("GeForce 9600GT GPU running times",600,400)
c1.visible()
c1.setNameY("Running time (milliseconds)")
c1.setNameX("Problem size (number of nodes)")
c1.draw( "", (17,17,17,34,34,34,36,36,36,39,39,39,
         43,43,43,45,45,45,48,48,48,65,65,65,
         70,70,70,71,71,71,100,100,100,323,323,323
         ),
         (3,3,3,6,6,10,11,11,11,13,7,7,
         7,7,7,23,9,24,18,27,35,46,45,46,
         35,35,35,36,36,37,245,211,317,16022,2955,18234) )
```

B.7 GPU Convergence Iterations on a 64-core card

```
from java.awt import Color,Font
from java.util import Random
from jhplot  import *
from jhplot.math.StatisticSample import randUniform

c1 = SPlot("GeForce 9600GT GPU convergence iterations",600,400)
c1.visible()
c1.setNameY("Convergence iteration")
c1.setNameX("Problem size (number of nodes)")
c1.draw( "", (17,17,17,34,34,34,36,36,36,39,39,39,43,43,43,
         45,45,45,48,48,48,65,65,65,70,70,70,
         71,71,71,100,100,100,
                   ),
         (1,1,1,1,2,1,2,2,2,2,1,1,1,1,1,
         3,1,3,2,3,4,3,3,3,2,2,2,
         2,2,2,7,6,9,43,8,49
         ) )
```

Appendix C

Original Project Proposal

Ruslan Kudubayev
Churchill College
rk379

Parallelising the Travelling Salesman Problem on NVIDIA®CUDA™architecture

19 Oct, 2009

Project Originator: Ruslan Kudubayev

Resources Required: See attached Project Resource Form

Project Supervisor: *Dr Jagdish Modi*

Signature:

Director of Studies: *Dr John Fawcett*

Signature:

Overseers: *Graham Titmus* and *Alastair Beresford*

Signatures:

Introduction and Description of the Work

Massively parallel computation has for a long time been considered expensive and was often not affordable to ordinary people interested in the subject. Nowadays, we are faced with a fast-paced market of supercomputing, so the general trend in the industry is performing massively parallel computation for a small price. One such example we are able to observe is the emergence of Graphical Processing Units(GPUs) produced by various companies.

GPUs are capable of producing some high-end graphics. However what is becoming more interesting now is general-purpose computation on GPUs, which allows very expensive real-world computation tasks to be solved on a relatively cheap graphics hardware. We particularly saw some interesting developments in scientific computations using GPUs, such as n-body simulations or monte-carlo simulations.

On the other hand, coming from the discrete theoretical world, there is a beautiful problem that has been tackled for many decades called Travelling Salesman Problem (TSP). In short, the problem is to traverse all nodes in a given graph, while visiting each node only once, also minimising the total weight of edges used in the traversal. This problem is well-studied and is known to be NP-Complete. There were some attempts at parallelising it, but none were observed for GPUs so far.

The purpose of this project is to study parallelisation of TSP on modern GPUs, in particular on modern NVIDIA graphics cards using the CUDA parallel computing architecture.

Extensions

One particularly interesting extension to this project would be to implement TSP using a MapReduce framework. MapReduce was introduced by Google and is very good at tackling distributed problems. There were some interesting attempts at trying to get MapReduce suited for CUDA framework and I want to compare the TSP solving MapReduce performance to the results achieved by GPUs and CPUs.

Resources required

Programming environment: NVIDIA CUDA SDK which is freely available on the NVIDIA website. The code used is a variant of C, modified to reflect architectural features like shared/device memory.
Hardware: This project obviously requires some NVIDIA graphics hardware. I own an Apple Macbook(2.13 GHz, 3MB L2 cache, 2GB main memory at 800MHz, 160GB hard drive) which has NVIDIA GeForce 9400M graphics card in it. It has 16 CUDA cores and will be sufficient for the aims of this project.

In an unfortunate case of my laptop breaking I will perform testing on either 1) my new laptop, 2) my friend's laptop (all Apple laptops are shipped with CUDA-enabled cards), 3) any computer by running special hardware emulation mode.
Disk space: This project does not require much disk space, so I will use my own computer to store files and will back them up regularly to the PWF systems. I will also use Subversion for version control which would be hosted at svnrepository.com (a paid service which gets backed-up very regularly).

Starting Point

I have some basic knowledge of using the CUDA SDK. In particular, so far I know how to multiply matrices in parallel and retrieve results. Experience gained on PartIA and PartIB will also be applied in some parts.

Substance and Structure of the Project

The aim of the project is to investigate the capabilities of GPU computing when applied to solving TSP. The developed testing environment is not intended to be suitable for general-purpose usage and is merely for extracting statistical data from experiments. Another substantial part of the project is comparison to CPU performance.

The project consists of the following main sections:

1. Research into different TSP algorithms both parallel and non-parallel and identify the most interesting ones to be compared. Come up with an efficient TSP algorithm which could extract the best from the hardware. Design data structures to be used for the algorithm chosen. Learn how to efficiently use the C-like programming environment of CUDA SDK provided by NVIDIA.

2. Implement TSP for both GPU and CPU.

3. Develop a convenient way of testing the outcomes for correctness. This would involve creating a special test generator which would generate random graphs as well as specialised tricky cases. I'm also planning to use some publicly available testsets of TSP that some researchers publish online.

4. Optionally arrange testing for different NVIDIA graphics cards with other departments or otherwise different parties.

5. Compare results for different approaches and analyse scalability/performance/memory footprints, etc.

6. Write the dissertation.

Success Criteria

The following should be achieved in order for the project to be considered successful:

- Implement an algorithm solving TSP which is a fairly substantial dataset problem on GPU. The implementation has to be either optimal or approximated, depending on the algorithm chosen.
- Evaluate performance depending on the number of cores used. Hence analyse scalability of the algorithm. It is expected that speed-up would not be linear in the number of cores, but still would show some considerable improvement as more cores are added.
- Compare GPU performance to CPU performance. Clearly, GPU should outperform CPU in time spent solving testsets.

Timetable and Milestones

Week 1 starts from 30/10/09 (the week after handing the project proposal). The dissertation is to be submitted on 14/05/10 which is week 28.

Weeks 1 and 2 (30/10/09 - 12/11/09

Study the TSP in detail. In particular, some attributes of the problem that are especially challenging to parallelise. Should also look at standard results from the Complexity Theory and classes of Parallel Complexity problems. Decide on how to implement the solution. One approach is to translate the problem into the world of matrix multiplications (which can be done very efficiently). Another approach is to study the problem from scratch by analysing data dependencies for a typical implementation. Reading up on TSP from distinguished research papers on the subject and also learn some special tricks about programming with CUDA to get the best performance.

Milestone 1: Understand the nature of the problem difficulties and complexities of parallelising it. Have a draft pseudocode of the algorithm I chose to use.

Weeks 3 and 4 (13/11/09 - 26/11/09)

Further theory study and negotiations with Project Supervisor to ensure that chosen direction is the right way to go. Trying out simple implementations of the algorithm chosen. Decide on a suitable C-implementation on CPU and discuss it with the Project Supervisor.

Milestone 2: Gained more knowledge to demonstrate more relevant results or possibly even partial problem solver on GPU. Come up with a draft pseudocode solving TSP on CPU.

Week 5 (27/11/09 - 3/12/09)

Aim to come up with more or less working version of TSP solver working on GPU.

Milestone 3: Have both GPU and CPU implementations working, and some test data ready as well.

End of Michaelmas term.

Weeks 6 to 10 (4/12/09 - 7/01/10)

Developing a CPU version of TSP solver. In parallel to that, developing a test set to use for testing correctness. That would be a test-set generator program (written in C) which would generate a substantial amount of test cases having graphs of different forms. Particularly interested in random sparse graphs and narrow specialised tricky graphs.

Start of Lent term.

Weeks 11 to 16 (8/01/10 - 18/02/10)

Starting to gather statistical data about the executions for different execution environments for both GPU and CPU. Starting writing up dissertation.

Milestone 4: Have most of the data for evaluation ready to be analysed. Start the Preparation chapter.

Weeks 17 to 20 (19/02/10 - 18/03/10)

Write a substantial part of the dissertation including any results achieved. Aiming to have evaluation part done to be reviewed with Project Supervisor in order to draw trends and reasons for them to arise. Carefully filling in the gaps in the dissertation based on the feedback received. Code should be pretty much polished and finalised now having been though many modifications arising from the results obtained.

Milestone 5: Implementation chapter mostly complete and Evaluation chapter reasonably prepared. Code working to produce right answers for the test sets provided.

End of Lent term.

Weeks 21 to 25 (19/03/10 - 22/04/10)

Slowing down project development with examination preparations taking up more priority. Still working on polishing the Implementation and Evaluation sections. If the project is on schedule, then work on extensions, otherwise use this period as slack time. Finish Conclusion chapter as well.

Milestone 6: Conclusion chapter completed.

Start of Easter term.

Weeks 26 to 28 (23/04/10 - 13/05/10)

Finishing with the dissertation and slack time.

Milestone 7: Dissertation ready to be handed in.