Generic Techniques in General Purpose GPU Programming with Applications to Ant Colony and Image Processing Algorithms

DAWSON, LAURENCE,JAMES

GENERIC TECHNIQUES IN GENERAL PURPOSE GPU PROGRAMMING WITH APPLICATIONS TO ANT COLONY AND IMAGE PROCESSING ALGORITHMS

Laurence Dawson

A thesis presented for the degree of

Doctor of Philosophy

School of Engineering and Computing Sciences

Durham University

July 2015

Abstract

In 2006 NVIDIA introduced a new unified GPU architecture facilitating general-purpose computation on the GPU. The following year NVIDIA introduced CUDA, a parallel programming architecture for developing general purpose applications for direct execution on the new unified GPU. CUDA exposes the GPU's massively parallel architecture of the GPU so that parallel code can be written to execute much faster than its sequential counterpart. Although CUDA abstracts the underlying architecture, fully utilising and scheduling the GPU is non-trivial and has given rise to a new active area of research. Due to the inherent complexities pertaining to GPU development, in this thesis we explore and find efficient parallel mappings of existing and new parallel algorithms on the GPU using NVIDIA CUDA. We place particular emphasis on metaheuristics, image processing and designing reusable techniques and mappings that can be applied to other problems and domains.

We begin by focusing on Ant Colony Optimisation (ACO), a nature inspired heuristic approach for solving optimisation problems. We present a versatile improved data-parallel approach for solving the Travelling Salesman Problem using ACO resulting in significant speedups. By extending our initial work, we show how existing mappings of ACO on the GPU are unable to compete against their sequential counterpart when common CPU optimisation strategies are employed and detail three distinct candidate set parallelisation strategies for execution on the GPU. By further extending our data-parallel approach we present the first implementation of an ACO-based edge detection algorithm on the GPU to reduce the execution time and improve the viability of ACO-based edge detection. We finish by presenting a new color edge detection technique using the volume of a pixel in the HSI color space along with a parallel GPU implementation that is able to withstand greater levels of noise than existing algorithms.

Declaration

The work presented in this thesis is based on research carried out at the Department of Engineering and Computing Sciences, Durham University, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Preface

The contents of this thesis have appeared, in part, in the following peer reviewed publications:

- L. Dawson and I. A. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA," in *IEEE Congress on Evolutionary Computation, IEEE-CEC'13, Cancun, Mexico, June 20-23, 2013*, pp. 1901–1908, 2013 [1]
- L. Dawson and I. A. Stewart, "Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU," in *Algorithms and Architectures for Parallel Processing*, vol. 8285 of *Lecture Notes in Computer Science*, pp. 216–225, 2013 [2]
- L. Dawson and I. A. Stewart, "Accelerating Ant Colony Optimization based Edge Detection on the GPU using CUDA," in *IEEE Congress on Evolutionary Computation, IEEE-CEC'14, Beijing, China, July 6-11, 2014*, pp. 1901–1908, 2014 [3]
- J. Zhao, Y. Xiang, L. Dawson, and I. A. Stewart, "Color Image Edge Detection based on Quantity of Color Information and its Implementation on the GPU," in *23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'11)*, pp. 116–123, 2011 [4]

Acknowledgments

I would like to thank my supervisor Professor Iain A. Stewart for all of his help, guidance and support throughout my undergraduate and postgraduate studies. There has rarely been an occasion over the last 4 years where he wasn't able to help solve even the most complex of problems with little more than a whiteboard and pen.

I would also like to thank Dr Yonghong Xiang for helping me settle in during the first year of my Ph.D and collaborating with me on my first paper.

Finally I would like to thank Durham University, the Department of Engineering and Computing Sciences and EPSRC for their financial support.

Contents

1 Introduction **15**

1.1 Graphics Processing Units 15

1.2 NVIDIA CUDA 16

1.3 Motivation 19

1.3.1 Metaheuristics 19

1.3.2 Image Processing 20

1.4 Contributions & Thesis Guide 20

2 General-purpose computation on the GPU using CUDA **23**

2.1 Introduction 23

2.2 CUDA hardware 24

2.3 The CUDA programming model 26

2.3.1 Blocks and Threads 26

2.3.2 Memory types 29

2.3.3 Synchronisation 35

2.3.4 OpenCL 38

2.4 Recent advances 39

2.4.1 Fermi, Kepler and Maxwell 39

2.4.2 Embedded and low power applications 42

2.4.3 CUDA roadmap 43

3 Parallel Ant Colony Optimization on the GPU **44**

3.1 Ant Colony Optimisation 44

3.2 Solving the Travelling Salesman Problem 45

3.2.1 Solution Construction 47

3.2.2 Pheromone Update 47
3.3 Parallel Ant Colony Optimisation 49
3.3.1 Parallel ACO on the GPU 49
3.4 Related work 50
3.4.1 Large TSP instances 51
3.4.2 Data-parallelism 52
3.5 Implementation 54
3.5.1 Tour construction 54
3.5.2 Double-Spin Roulette 57
3.5.3 Pheromone update 60
3.6 Results 61
3.6.1 Experimental Setup 61
3.6.2 Solution Quality 61
3.6.3 Benchmarks 62
3.6.4 Shared memory restrictions 65
3.7 Conclusion 66
3.7.1 Contributions 66

4 Candidate Set Parallelisation Strategies 67
4.1 Introduction 67
4.2 Candidate Sets 68
4.2.1 Using candidate sets with the TSP 69
4.2.2 The effect of utilising a candidate set 70
4.3 Related work 72
4.4 Implementation 73
4.4.1 Basic setup 73
4.4.2 Candidate set setup 73
4.4.3 Tour construction using a candidate set 74
4.4.4 Task parallelism 75
4.4.5 Data parallelism 75
4.4.6 Data parallelism with tabu list compression 80
4.5 Results 81

4.5.1 Experimental Setup 81
4.5.2 Benchmarks 81
4.6 Conclusion 83
4.6.1 Contributions 84

5 Ant Colony Optimization based Image Edge Detection 85
5.1 Introduction 85
5.2 Edge detection using ACO 86
5.2.1 Solution construction 87
5.2.2 Pheromone update 88
5.2.3 Termination conditions 88
5.3 Related work 89
5.3.1 Edge detection using Ant Colony Optimisation 89
5.3.2 Other edge detection methods on the GPU 90
5.4 Implementation 90
5.4.1 Adapting data-parallelism for edge detection 90
5.4.2 Algorithm setup 91
5.4.3 Solution construction 92
5.4.4 Pheromone update 95
5.5 Results 96
5.5.1 Experimental setup 96
5.5.2 Algorithm parameters 96
5.5.3 Solution quality 97
5.5.4 Variable edge thickness 98
5.5.5 Benchmarks 99
5.6 Conclusion 100

6 Color Image Edge Detection 101
6.1 Introduction 101
6.2 Color image edge detection 102
6.2.1 Edge detection techniques 102
6.3 Related work 103
6.4 Implementation 104

6.5 Parallel Implementation on the GPU 106
6.5.1 Algorithm setup 106
6.5.2 Edge detection 107
6.6 Results 110
6.6.1 Noise tolerance 111
6.6.2 Improved edge detection 112
6.6.3 GPU Implementation Results 113
6.6.4 Experimental setup 113
6.6.5 Benchmarks 113
6.7 Conclusion 114

7 Conclusion **115**
7.0.1 Future work 116

Bibliography **117**

List of Figures

2.1 A comparison of the floating point performance. 25
2.2 A two-dimensional arrangement of 8 thread blocks within a grid. . . 26
2.3 An overview of the hierarchical CUDA memory model. 29
2.4 An example of multiple threads across separate thread warps executing the CUDA all warp vote function and following different execution paths. 32
2.5 An example of all threads within a block accessing the same 128 byte region of global memory resulting in a single 128 byte load. 34
2.6 An example of the threads from the same block accessing two 128 byte regions of memory resulting in two separate global memory loads. 34
2.7 An example of global thread block synchronisation via executing the function cudaDeviceSynchronize() to explicitly synchronise all blocks. 36

3.1 An overview of the Ant System algorithm. [19] 46
3.2 An example of performing the 2-opt operation on a tour. The edges between (c, d) and (h, g) and are replaced with new edges between (c, g) and (d, h). 46
3.3 Overview of an ant's tour construction. 55
3.4 An overview of the double-spin roulette algorithm executed by each ant (thread block) during each iteration of the tour construction phase of the AS algorithm in order to construct a valid tour. 58
3.5 The warp-reduce method [52]. 59
3.6 A comparison of the quality of tours constructed via the existing CPU implementation of AS and new GPU implementation. 62

3.7 Speedup of execution against the standard CPU implementation. . 64
3.8 Speedup of execution against the best existing GPU implementation. 64
3.9 Observed execution speeds as a product of varying the L1 cache size. 66

4.1 Performance of the GPU implementation of AS presented by Cecilia et al. [20] without the use of a candidate set against the sequential CPU implementation [53] using a candidate set. 71
4.2 Performance of our GPU implementation of AS without the use of a candidate set against the sequential CPU implementation [53] using a candidate set. 71
4.3 Overview of an ant's tour construction when using a candidate set. 74
4.4 An overview of our data parallel tour construction algorithm utilising a candidate set to reduce the total execution time of the algorithm. 79
4.5 A comparison of the speedup of execution of multiple GPU instances (with and without use of a candidate set) against the standard CPU implementation using when using a candidate set. 82

5.1 Surrounding neighbour pixels and valid moves from position (i, j) (providing none of the surrounding pixels have recently been visited) 88
5.2 Calculating the greyscale value for a color pixel 91
5.3 An overview of our novel mapping technique to pack multiple ants within a thread block. Each ant is mapped to a thread warp and four warps are executed per thread block in parallel. 93
5.4 An overview of solution construction 94
5.5 A comparison of final edge maps produced by the Sobel, Canny and our parallel ACO edge detection algorithms. 97
5.6 The effect on edge thickness when alternating the number of ants in a 512x512 standard test image. 98
5.7 Observed execution speeds for the ant-to-warp mapping with/without caching visited positions to shared memory. 100

6.1 Load the top apron pixels 109
6.2 Load the centre and apron pixels 109

6.3 Load the bottom apron pixels 109
6.4 LUT index matrix 110
6.5 The effect of introducing noise to the input image when using the standard image processing test image *Lena*. 111
6.6 Edge maps produced for a color image. 112

List of Tables

2.1 A comparison of consumer level CUDA GPUs for each hardware revision. 39

3.1 An example run of roulette wheel selection (proportionate selection). 56
3.2 The best tour lengths obtained for various test instances of the TSP on the CPU and GPU implementations over 10 independent runs with 100 iterations. 62
3.3 The Average execution times (ms) of our GPU implementation against the best current GPU implementation from Cecilia et al. and the standard sequential test CPU implementation (ACOTSP [53]). . . . 63
3.4 The execution times for each stage of our proposed GPU implementation (ms) across various instances of the TSP. 63

4.1 The average execution times (ms) of the CPU implementation (with and without candidate set), the best current GPU implementation from Cecilia et al. and our GPU implementation. 70
4.2 Average execution times (ms) when using AS and a candidate set. . 82

5.1 Average execution times (ms) when varying the number of threads per block and mapping arrangement of ants to blocks, threads and warps. 99

6.1 Average execution times (ms) for processing standard image processing test input images using the OpenCV canny implementation, the CUDA GPU implementation [78] with 4 passes of the hysteresis step, the CUDA GPU implementation [78] with 10 passes of the hysteresis step and finally using our new color edge detection method. 113

Glossary

$\mathcal{MAX} - \mathcal{MIN}$ Ant System.

ACO Ant Colony Optimization.

AS Ant System.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

FPS Frames per Second.

GPGPU General-Purpose computing on Graphics Processing Units.

GPU Graphics Processing Unit.

HSI Hue-Saturation-Intensity.

IPP Intel Performance Primitives.

LUT Lookup table.

NPP NVIDIA Performance Primitives.

OpenCL Open Computer Language.

PCB Printed Circuit Board.

SM Streaming Multiprocessor.

SoC System on a chip.

SP Stream Processor.

SPMD Single-program, multiple-data.

STL Standard Template Library.

thread warp A collection of 16 to 32 CUDA threads executed in parallel within a thread block. A thread block can consist of multiple threads warps.

TSP Travelling Salesman Problem.

CHAPTER 1

Introduction

1.1 Graphics Processing Units

In November 2006 NVIDIA introduced the GeForce 8 Series architecture along with the next generation of *Graphics Processing Units* (GPUs). This generation pioneered the unified shader pipeline combining pixel and shader units into unified shaders each with their own instruction memory, cache and control logic [5]. This move to a unified architecture resulted in each of the units becoming fully programmable and no longer constrained to a single graphics task. Each of the unified shader units execute independently in parallel allowing the GPU to dynamically allocate execution resources depending on the workload. These changes in turn led to increased performance and efficiency [6]. Rege notes [7] that by moving to a unified architecture we can achieve up to a 2x increase in instruction throughput by using scalar instruction shaders resulting in significantly better performing complex graphic shaders (on the first generation of hardware alone). The resulting GPU architecture was a massively parallel general-purpose processor with large memory bandwidth and improved floating point performance that enabled both traditional graphics processing and, more importantly, facilitated general-purpose computation outside of the realm of graphics with the aid of the new unified shader units. These fundamental changes to create a new unified general-purpose GPU architecture were implemented in consumer grade GPUs available in standard desktops and laptops, so putting a parallel processor within reach of all users

Over the last decade parallel processing has become increasingly popular due in part to the diminishing returns when increasing the clock rate of conventional CPUs. Additional power consumption and cooling requirements outweighed the increased performance benefits after CPUs began to approach a clock speed of around 4 GHz [8]. This speed limit restricted the potential of single core CPUs leading to the rise of multiple core CPUs. A typical desktop computer is currently equipped with around 4 to 8 cores per CPU and this can increase upwards of 12 cores for higher end workstations or servers depending on the configuration.

1.2 NVIDIA CUDA

In June 2007 NVIDIA unveiled Compute Unified Device Architecture (CUDA), a parallel programming architecture that allowed developers to harness the massively parallel architecture of the new unified GPU for general-purpose computation (GPGPU). CUDA was the first language designed to facilitate GPGPU using ANSI C which required no prior experience with OpenGL or DirectX. This gave CUDA a competitive advantage over previous attempts which forced developers to coerce non-graphics algorithms into restrictive graphics interfaces for execution on the GPU. By facilitating execution on the massively parallel architecture, code can be written to execute in parallel much faster than its optimised sequential counterpart. Parallel processing allows work to be divided into smaller units and executed in parallel as opposed to executing sequentially. NVIDA note [9] that to get the maximum benefit from CUDA it's best to first find efficient ways to [9] sequential code. By applying Amdahl's Law [9] we can determine the maximum expected speedup from moving sections of sequential code to execute in parallel. NVIDIA note [9] that to maximise the speedup it's worthwhile spending effort increasing larger regions of code to be parallelised. CUDA compatible GPUs can be found in a wide range of devices from desktop computers to laptops and more recently mobile devices such as tablet computers and phones [10]. This allows developers to utilise massively parallel processing in previously restricted domains and crucially improve the performance of their applications whilst preserving portability due to the prevalence of NVIDIA GPUs. Additionally CUDA applications can scale at

runtime to utilise all available processing units so as to achieve the best possible runtime on the available hardware.

Since 2007 finding efficient CUDA implementations and parallel mappings of selected algorithms has been an active and ongoing area of research. There have been seven major releases of the CUDA toolkit along with four major hardware revisions; Tesla, Fermi, Kepler and Maxwell (see Section 2.4.1). In 2008 Tokyo Tech announced [11] the first GPU powered supercomputer to enter the list of top 500 supercomputers worldwide using GPGPU to achieve this. The following year NVIDIA released the first major hardware revision to the platform (Fermi) which significantly improved the speed of the GPU but also releaxed some of the tight memory coalescing restrictions (see Section 2.3.2) imposed by the first hardware generation vastly increasing the programmability of the platform. In 2011 the National Supercomputing Center announced [12] the world's fastest supercomputer Tianhe-1A based on NVIDIA GPUs. The Tianhe-1A used a total of 7168 NVIDIA Tesla GPUs to run molecular dynamics simulations at 1.87 petaflops per second [12]. In 2012 NVIDIA announced the Kepler hardware architecture bringing further CUDA advancements including dynamic parallelism (see Section 2.4.1). In 2012 NVIDIA also announced a virtualised GPU cloud solution known as GRID [13] allowing developers to access the power of the GPU on cloud based platforms such as Amazon Web Services (AWS). In 2014 NVIDIA released the most recent hardware revision of CUDA Maxwell [14]. Maxwell brought continued speed improvements along with an additional energy efficiency. CUDA has been successfully applied to a wide range of problems across multiple domains including medical imaging [15], computational finance [16], bioinformatics [17] and many other areas. CUDA success stories tout significant execution speedups over optimised sequential counterparts, each improving the viability of general-purpose GPU computing.

The simplest parallel implementations can easily offload entire computationally expensive regions of an application to the GPU using CUDA. However this method can require costly memory transfers to and from the GPU which can increase the execution time resulting in marginal speedups when compared to sequential counterparts. This simple implementation only requires minimal effort to port regions of an application as opposed to redesigning an entire application to execute

in parallel. Libraries such as Thrust [18] provide high level interfaces based on the Standard Template Library (STL) to perform common methods such as sorting without requiring detailed knowledge of CUDA programming.

Although CUDA abstracts the underlying GPU architecture, fully utilising and scheduling the GPU is non-trivial. Along with a hierarchical memory system, CUDA imposes many restrictive limitations which must be strictly adhered to in order to attain maximum speedups and performance. As a result, many existing sequential and even parallel algorithms must be redesigned in order to conform to the restrictive parallel GPU architecture. To attain the best performance it is often necessary that the entire application is executed in parallel on the GPU to avoid the aforementioned memory transfers. This can result in a significantly more complex implementation, increased development time and unforeseen parallelisation issues. The complexity of implementing existing algorithms on the GPU depends heavily on the resources used, interaction between processes and control flow of the algorithm. Some algorithms (such as basic image processing) can be classified as *embarrassingly parallel* when little effort is required to split up the process into parallel tasks and are often easily ported to the GPU. Executing in parallel can also bring unexpected scheduling difficulties as some algorithms can be highly sequential thus prohibiting parallelisation and greatly increasing the effort required to implement an efficient CUDA solution that executes faster than a optimised sequential counterpart. In the worst case, regions of algorithms that cannot be easily executed in parallel must still be executed on the GPU to avoid the time penalty of memory transfers. NVIDIA provide a set of comprehensive best practices to help developers port and implement existing and new algorithms on the GPU using CUDA [9]. However, deviating from these strictly defined best practices can lead to severe performance issues such as warp serialisation which imposes a harsh time penalty by executing regions of parallel code sequentially, often negating the performance benefit of executing in parallel.

In summary, developing an efficient parallel implementation is highly dependent on the input algorithm. Although a simple parallel implementation can be easily implemented, the benefits of executing in parallel are often hindered by poor performance. Some problems can be easily parallelised and others require significant

changes to adapt sequential regions of the algorithm to execute in parallel without scheduling issues. For maximum speedups all features of the GPU must be utilised requiring detailed knowledge of the GPU however; this is not always possible.

1.3 Motivation

Due to the aforementioned complexities pertaining to GPU development, the motivation behind this thesis is to explore and find efficient parallel mappings of existing and new algorithms on the GPU using NVIDIA CUDA. We place particular emphasis on metaheuristics, image processing and designing reusable techniques and mappings that can be applied to other problems and domains. By extending recent contributions we start by focusing on Ant Colony Optimisation (ACO) and move on to new parallel applications of ACO and image processing.

1.3.1 Metaheuristics

Ant Colony Optimisation is a population-based metaheuristic that has proven to be the most successful ant algorithm for modelling discrete optimisation problems including the Travelling Salesman Problem (TSP) [19]. However, when modelling the TSP, as the number of cities to visit increases, so does the computational time required to construct tours. This problem necessitates the need for a parallel implementation in order to reduce the execution time and increase performance. Many of the nature inspired heuristic approaches for solving optimisation problems may seem intrinsically and even embarrassingly parallel. Dorigo and Stützle remark [19] that for many applications ACO solutions often rival the best in class, however, as previously mentioned, finding efficient parallel implementations to exploit the GPU hardware is non-trivial. Recent contributions from Cecilia et al. [20] and Delévacq et al. [21] have shown that adopting a data-parallel approach can better utilise the GPU. However the speedups attained are relatively low, and both implementations fail to utilise CPU optimisations and only apply their solution to the TSP. Motivated by the high quality of solutions produced by ACO and the recent parallel GPU developments we focuss on extending and improving data-parallel techniques whilst utilising CPU optimisations in parallel.

1.3.2 Image Processing

Edge detection is one of the most fundamental operations in computer vision and image processing as other tasks (such as image segmentation, object recognition and classification) can depend upon edge characterisation. It is crucial that the process of edge detection should result in a precise characterisation of the image features. Hence, edge detection must be both reliable and efficient. Edge detection differs according to whether an image is color or not. Novak and Shafer [22] found that about 90% of edges in color images are also edges in terms of their gray values. However, the remaining 10% of edges in color images can not be so characterised. Thus, many color edge detection algorithms have been proposed (see [23, 24, 25] for examples). In spite of this, there has been little research into developing efficient parallel implementations aside from traditional algorithms such as the Canny edge detection algorithm [26]. Luo and Duraiswami [27] and Ogawa et al. [28] both present GPU implementations of the Canny algorithm. Luo and Duraiswami [27] note that over 75% of the runtime is spent applying edge thresholding to improve the quaility of edge maps. However, neither implementation considers using color images to improve the quality of edges detected.

1.4 Contributions & Thesis Guide

As previously mentioned this thesis focuses on finding efficient parallel mappings and optimisations of the Ant Colony Optimisation metaheurisitc and image processing algorithms on the GPU using NVIDIA CUDA and is based on the publications [1, 2, 3, 4]. We omit a single related work section in favour of providing detailed related sections. Our primary contributions made and guide to this thesis are outlined as follows:

- In Chapter 2 we provide additional background on parallel GPU development and give an overview of the hardware of the GPU, the CUDA programming model and recent CUDA hardware advancements.
- In Chapter 3 we present an improved data-parallel approach for mapping the Ant System algorithm to the GPU for solving the Travelling Salesman

Problem. We detail a novel implementation of a parallel warp-level roulette wheel selection algorithm called Double-Spin Roulette which is able to significantly outperform the execution time of existing parallel and sequential contributions whilst matching the quality of sequentially generated solutions. Our results show a speedup of up to 8.5x faster than the best existing GPU implementation and up to 82x faster than the sequential counterpart.

- In Chapter 4 we show how existing mappings of ACO on the GPU are unable to compete against their sequential counterpart when common optimisation strategies such as the use of candidate sets are employed. We explore three distinct candidate set parallelisation strategies for execution on the GPU. We show that by extending our previous data-parallel and warp-level approach we were able to efficiently integrate the use of a candidate set to reduce the overall execution time against both sequential and parallel counterparts. Our results show a speedup of up to 18x faster than the sequential counterpart while the best existing implementations struggle to maintain a speedup

- In Chapter 5 we present the first implementation of a parallel ACO-based edge detection algorithm on the GPU using NVIDIA CUDA. By further extending our data-parallel and warp-level approach we map individual ants to thread warps allowing multiple ants to execute on each CUDA block thus exploiting the GPU hardware and was able to yield the fastest execution times. Consequently, we increase the viability of ACO-based edge detection. Our approach is also able to produce variable edge widths that can produce stylised edge maps. Our results show a speedup of up to 150x faster than the sequential counterpart

- In Chapter 6 we propose a new method for quantifying color information so as to detect edges in color images using the volume of a pixel in the HSI color space. We detail a novel GPU implementation featuring an efficient parallel edge thinning algorithm that precomputes all potential thinning outcomes based on surrounding pixel data and storing the results in a look up table. We show that our method can improve the accuracy of detection, withstand greater levels of noise in images and decrease the execution time thus attaining

a speedup against related implementations. When comparing the execution times of our new color edge detection algorithm against the Canny edge detection algorithm our results show a speedup of up to 5x faster than the best performing GPU implementation and up to 20x faster than the sequential counterpart.

- In Chapter 7 we conclude this thesis and give direction for potential future work based on our research and findings.

CHAPTER 2

General-purpose computation on the GPU using CUDA

2.1 Introduction

In Chapter 1.1 we introduced the modern unified GPU and outlined how recent advancements have facilitated the development of the next generation of general-purpose computation on the GPU (GPGPU). By fully harnessing the power of a massively parallel processor, developers can significantly improve the performance of their applications on low-cost GPU hardware now available in most laptops, desktops and recently even mobile devices (including tablet computers). CUDA has since been applied to a wide range of problems across multiple domains.

In Chapter 1.2 we introduced CUDA, a parallel architecture designed for executing applications in parallel on the new unified GPU architecture. In this thesis we focus on finding efficient parallel mappings of existing algorithms to the massively parallel GPU using CUDA. It is important to fully understand the architecture of the modern massively parallel GPU and CUDA programming model before further discussing our primary contributions. In this chapter we provide background information on the GPU and CUDA detailing: the hardware of a CUDA compatible GPU, the CUDA programming model including how it must be utilised when developing parallel applications and the considerations that must be taken into account when developing for parallel execution on the GPU, and recent advancements made to the GPU hardware, the CUDA software toolkit and support for new processor architectures (notably ARM).

2.2 CUDA hardware

First and foremost a CUDA compatible GPU is a separate processing unit that runs alongside the CPU. A modern desktop computer may be equipped with a single GPU or multiple GPUs depending on the configuration. A high-end desktop NVIDIA GPU can be purchased for around £500 and easily added to a standard motherboard. This enables developers to conveniently upgrade the GPU or add GPU support irrespective of the rest of their system. It is important to note that the GPU runs asynchronously to the CPU and therefore work can be distributed between the CPU and GPU [29]. Work can either be executed on the GPU blocking the CPU or executed on the GPU and asynchronously notify the CPU the work has completed. Typically an application will use one of two methods when scheduling work to the GPU. The first method relies upon a high level of parallelism in an application and uses the GPU alongside the CPU in parallel. The second method is suited to an application where events must strictly execute sequentially and each step is dependent on the previous. A computationally expensive region of the application is executed on the GPU, halting execution on the CPU, until the GPU has completed the unit of work. Both methods highlight the concept of using the GPU as a coprocessor alongside the CPU.

Since the introduction of the unified GPU, the typical architecture of a CUDA-compatible GPU consists of a scalable array of threaded streaming multiprocessors (SM), each containing a subset of stream processors (SP) that share control and instruction logic [5]. The number of SMs varies between devices with low-end devices having far fewer than high-end devices and CUDA applications scale transparently to the number of SMs available. The combination of multiple SMs form a CUDA block and the number of SMs required to constitute a block varies between each generation of CUDA. Each SP can also be known as a CUDA core and the total number of CUDA cores per device has increased significantly since the launch of the first generation of CUDA GPU. For example, the first flagship CUDA GPU (8800GTX) was equipped with 128 CUDA cores and this has risen to 2688 CUDA cores for the latest flagship CUDA GPU (Titan). From this we can see that the massively parallel architecture has increased over time with more CUDA cores

now available for parallel execution. There has also been considerable continued improvements to the clock rate, available memory, memory speed and bandwidth. In Fig. 2.1 we show the increase in floating point performance over the last 10 years on Intel CPUs and consumer level GeForce GPUs [30]. NVIDIA note that the dramatic increase in floating point performance on the GPU is due to the GPU specialising in graphics and therefore highly parallel intensive computation. For this reason the GPU assigns more transistors to data processing [30]. The major jumps in GPU performance of single and double precision data correspond to each major GPU hardware release for example from Fermi to Keppler (580 to 680)

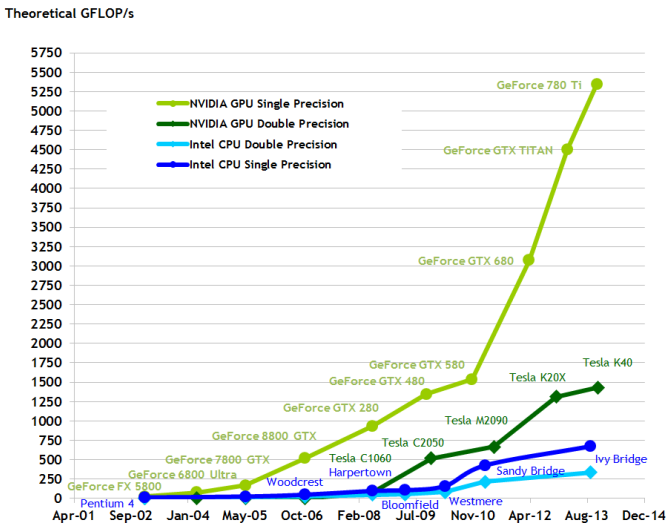


Figure 2.1: A comparison of the floating point performance.

Each GPU has access to a variety of memory types each possessing a range of different properties such as size, speed, location, volatility and built-in caching. In Section 2.3.2 we will discuss the implications this has on the software model and the affect on application performance. Each GPU has its own high-speed physical memory on the PCB of the GPU and the memory space of the GPU is separate to that of the CPU. Data available on the host (CPU) must be transferred to the device (GPU) as the device is unable to access the host memory space. The size of physical memory available on the PCB varies depending on the device and as expected high-end devices typically provide more memory. The current flagship Tesla GPU provides 6GB of high-speed GDDR5 memory.

2.3 The CUDA programming model

As previously mentioned, CUDA allows developers to execute methods in parallel directly on the GPU using a parallel programming interface. These parallel methods are known as *kernels* and are defined using the specifier *__global__* at the start of the method signature. When a kernel method is executed, the application moves execution from the host (CPU) to the device (GPU) [5]. NVIDIA note that unlike a conventional C function, a CUDA kernel is executed N times by N parallel CUDA threads [30]. It it important to note that kernels execute asynchronously and as a result kernels cannot return values. However, as expected, kernels can interact and write to memory but sequential execution must be blocked if subsequent stages depend on the results so as to avoid race condition synchronisation issues (see 2.3.3).

In the following subsections we will describe how kernels execute in parallel using the block and thread model, the memory types available for CUDA threads and execution synchronisation. Within each subsection we will also discuss common implementation issues and design considerations. We direct the reader to the NVIDIA programming guide for additional information regarding CUDA [30].

2.3.1 Blocks and Threads

When a kernel method is executed, the execution is distributed over a *grid* of *thread blocks* as illustrated below in Fig. 2.2. Each of the thread blocks contains a subset of parallel CUDA threads which all in turn execute the kernel method.

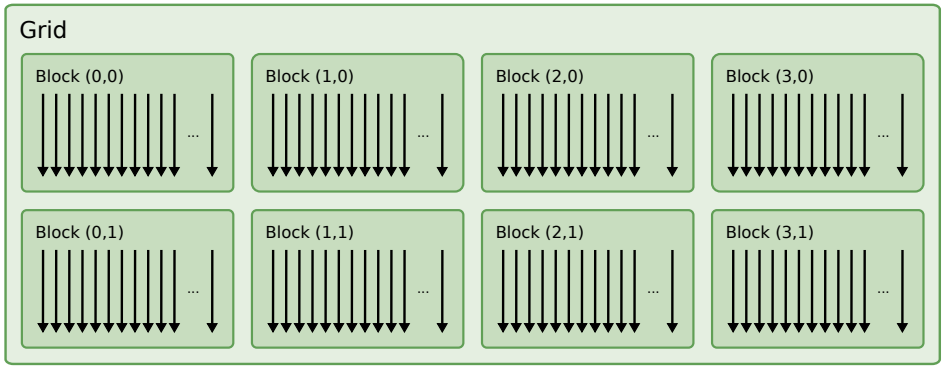


Figure 2.2: A two-dimensional arrangement of 8 thread blocks within a grid.

Blocks

Within the CUDA grid, thread blocks can be arranged into a one-dimensional, two-dimensional or three-dimensional formation [30] (an example of a two-dimensional arrangement can be seen in Fig. 2.2). Each thread block contains a number of parallel threads that can only directly interact with threads within the thread block. Each block has a unique position in the grid and this can be determined at run time by each thread. The number of thread blocks and their dimensional arrangement can be defined by the application before or after compilation and will vary depending on the usage. For example, when working with images, a 2-dimensional arrangement is best suited as this matches the two-dimensional array holding the pixel data (although a one-dimensional array could be used with additional calculations to determine the position of the block within the grid).

NVIDIA note [30] that thread blocks must be required to execute independently and this is key to the mapping of thread blocks to SMs. There can be no inferred or presumed order of execution and parallelism between thread blocks. As a result each block must be self contained. The number of thread blocks chosen for each kernel execution should be sufficiently high so as to utilise all of the parallel cores available. During execution we must assume that each block could be the first or the last block to execute. This execution independence allows blocks to be distributed across a variable number of processing elements and executed in parallel [30]. However, as multiple blocks can execute in parallel we must be aware of data integrity when accessing memory outside of the block thus necessitating the use of atomic operators (see 2.3.3) that respect the order of access. Farber notes [29] that NVIDIA's insistence on thread block independence might also be indicative of transparent scheduling execution across multiple devices in future releases.

Threads

Within each thread block is a collection of parallel threads. Older cards such as the G80 can support up to 768 parallel threads per block and more recent Fermi cards increased that limit to 1024. The total number of threads per kernel execution is equal to the number of threads per thread block multiplied by the

number of thread blocks. For example executing 128 thread blocks each with 1024 threads would result in 131072 threads each executing the same kernel. Each parallel thread is only able to communicate with other threads in that block and this is facilitated by *shared memory*. Within a block, threads execute in parallel in smaller sub-blocks known as warps. Each thread warp contains either 16 or 32 threads depending on the compute version and generation of CUDA GPU. There is no guarantee as to what order the warps will execute in; however, within a warp threads can communicate directly with each other using warp level primitives such as *__ballot()*. As with thread blocks, the number of threads per block is configurable at runtime but each thread block must have the same number of threads. As a general rule, CUDA applications must keep the GPU busy and achieve a high level of occupancy. NVIDIA recommend that at least 64 threads are used per thread block [30]. Threads can also be configured into a one-dimensional, two-dimensional or three-dimensional formation. The arrangement of threads within a thread block is user configurable and certain configurations may suit particular applications [30].

For optimum performance, all threads within a warp must follow the same execution path. Developers must pay attention to the control flow of each warp so as to avoid branching conditional statements. For example, *if statements* may cause the warp to branch as CUDA cannot process the resulting multiple paths from this branching statement in parallel. When threads follow different execution paths within a warp, the execution is serialised resulting in a loss of parallelism within the warp until after the branching region has finished execution. This process of warp serialisation due to branching is known as *warp divergence* and must be avoided if possible. To avoid warp divergence we must ensure that all threads strictly follow the same control flow throughout the kernel execution even if this results in performing redundant computation to align all threads within the warp.

The CUDA programming model allows applications to support fine-grained parallelism capable of instantiating hundreds of thousands of parallel threads for execution on a single task. In contrast CPU applications often apply coarse-grained parallelism typically focusing a significantly lower number of threads on distinctly different tasks. As each of the threads executes the same kernel, we can classify this parallel model of execution as single-program, multiple-data (SPMD) [5].

2.3.2 Memory types

CUDA exposes a set of different memory types to developers, each with unique properties that must be exploited in order to maximise performance [30]. Each CUDA thread has access to all of the different memory types but it is up to the developer to know which memory type is applicable for their particular application and how best to utilise each of the different types for performance gains. In Fig. 2.3 we present an overview of the hierarchical CUDA memory model highlighting each of the different available memory types and the accessibility of each memory type. We will now review each of the memory types, detailing their unique properties, accessibility, scope and implementation considerations.

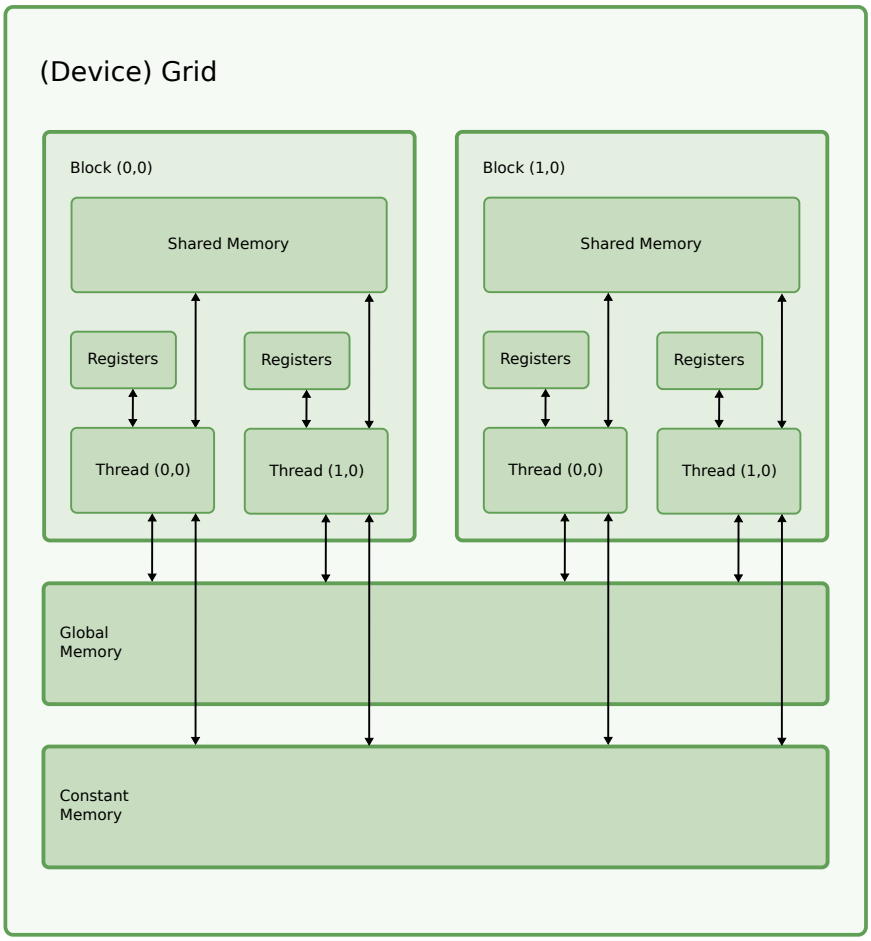


Figure 2.3: An overview of the hierarchical CUDA memory model.

Register Memory

The first type of memory is 32-bit *register memory*. Registers are the fastest form of storage and each thread within a block has access to a set of fast local registers that exist on-chip. However, the accessibility of register memory is severely limited as each thread can only access its own registers. As the number of registers per block is limited, blocks with many threads will have fewer registers per thread. If a thread block uses a high number of threads, each using multiple local variables, the block may use more registers than available. In this case the registers will *spill* into the significantly slower region of local memory resulting in loss of performance. During compilation the number of spilled registers is presented to the developer and care must be taken to keep this number to a minimum. A kernel may fail to execute if there are not enough registers or shared memory for at least one block to execute [30]. In short, registers are the fastest of all the available memory types but are in limited supply, must be managed to avoid register spilling and do not allow inter-thread communication facilitating the need for additional memory types.

Shared Memory

For inter-thread communication within a thread block, shared memory is the fastest available memory type and must be used to achieve optimum performance. As with register memory, shared memory exists on-chip but is accessible to all threads within the block. This increases the scope of accessibility from register memory and thus facilitates fast inter-thread communication within the thread block without moving to off-chip memory. Shared memory is slower than register memory and if a thread can execute without sharing information with other threads, shared memory should not be used over register memory as this will reduce the execution performance. When possible, shared memory should also be cached into local register memory to avoid additional memory reads (providing the data in shared memory is constant and using additional variables will not exhaust the number of registers available). Shared memory is extremely important for increasing application performance. When used correctly, data can be cached locally as to reduce the dependency on additional expensive reads to slower off-chip memory.

The amount of shared memory per block is limited to 16KB or 48KB depending on the split of memory with the L1 cache [29]. Shared memory is arranged into banks and developers must carefully manage access to these banks. On *Fermi* CUDA devices with compute version 2.0, shared memory is arranged as 32 consecutive 32 bit (4 byte) wide memory banks each with a bandwidth of 32 bits per cycle. This configuration is designed to permit each bank to load a single 4 byte word per clock cycle. For optimum performance, each thread in a warp (32 threads) must access a single 4 byte word from a different memory bank. If two or more threads attempt to access a single memory bank this is known as a *bank conflict* and all subsequent memory requests are serialised. To avoid serialisation, developers must explicitly manage all shared memory reads within a warp to access different memory banks. If just a single thread accesses the same bank as another thread within the warp, this will trigger a bank conflict. An exception to this rule is when all threads within a warp access the same word in a memory bank. In this case the value can be broadcast to all the threads within the warp using a single read operation opposed to multiple serialised reads [29]. Since compute version 3.0, the bank size is now adjustable and can be either set as 4 or 8 bytes. This allows developers to easily support double precision words without bank conflicts [30].

Inter-thread communication without shared memory

Aside from shared memory, threads can also communicate directly within a warp by using warp vote functions. The three available warp vote functions allow threads in a warp to communicate to other threads in the warp via a reduction-and-broadcast operation [30] and without using shared or global memory. The mechanism behind the warp vote is simple; each of 32 threads pass an integer into the function and the result is then broadcast to all threads in the warp. An example warp vote operation is the *all* function. The result of the *all* warp vote is only true if all of the 32 threads input a positive integer. In Fig. 2.4 we illustrate multiple threads across thread warps within a single thread block each executing the all warp vote function at the start of the kernel. In the first warp, the input into the vote function from each thread is positive and therefore the warp vote returns true. As a result all threads within the first warp follow the true execution path and so the warp

does not branch or trigger warp divergence. In the second warp the input contains non-positive integers and so the warp vote returns false. The warp then proceeds to follow the false execution path also without branching. The warp vote functions provide one of the most useful techniques to help reduce warp divergence and thus serialising the warp. Warp voting permits all threads in the same warp to check that a condition has been met before performing a potentially branching statement. By balloting threads within a warp, we can dynamically change the execution path depending on the result of the thread consensus reached via voting functions within the warp without using shared memory or serialising the warp.

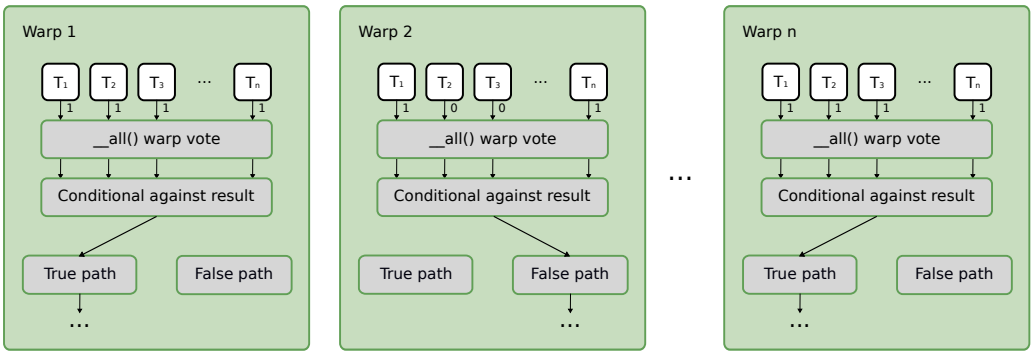


Figure 2.4: An example of multiple threads across separate thread warps executing the CUDA all warp vote function and following different execution paths.

As of compute version 3.0, newer *Kepler* GPUs can also communicate directly with other threads within their warp using the new warp shuffle functions [31]. The four available warp shuffle functions (*shuffle*, *shuffle up*, *shuffle down* and *shuffle xor*) allow single 4 byte variables to be transferred simultaneously between threads in each warp. As with warp vote functions, warp shuffle functions do not require the use of shared memory and only use one instruction to share variables as opposed to three [32]. The shuffle method must be executed twice sequentially when transferring larger 8 byte variables such as doubles. By using the shuffle function the amount of shared memory used per warp can be reduced which can result in a higher occupancy [32]. As each of the threads within the warp execute in parallel, there is no need to synchronise the warp after performing either a warp vote or warp shuffle thus further increasing the performance of the warp level operations.

Global, Constant and Texture Memory

For inter-block communication, larger data sets, and persistent memory, threads have access to *global (DRAM)*, *constant* and *texture memory*. The lifetime of global, constant and texture memory is the lifetime of the application (as opposed to register and shared memory which is only the duration of a single kernel execution). If an application requires memory persistence between multiple kernel executions, global, constant or texture memory must be used. From the device and within kernel execution, global memory has read and write access where as constant and texture memory only has read access. Constant and texture memory can only be written to by the host (CPU) prior to kernel execution. Since the release of compute version 2.0, access to global memory is now automatically cached using L1 and L2 caches. As previously mentioned, the amount of shared memory and the size of the L1 cache can be adjusted by the developer according to the use case and requirements [29]. For example, if each each thread block requires a large amount of shared memory the size of the L1 cache can be reduced to accommodate this. If global memory is accessed frequently (or randomly) the size of the L1 cache can be increased to hide the access latency by reducing the amount of shared memory per block. Texture and constant memory also benefit from caching, but the initial load will still be significantly slower than accessing shared or register memory.

Two of the CUDA implementation best practices detailed by NVIDIA [31] are to minimise redundant accesses to global memory and if possible to coalesce these global memory accesses. One common practice to minimise redundant access to global memory is to cache global memory locally in shared memory. A simple approach can instruct each thread to copy a value from global memory into shared memory. After the initial load all subsequent global memory read accesses would be directed to the shared memory array thus avoiding excessive costly global lookups. If required, any changes can be written from shared memory back into global memory at any point during the execution of a kernel. By replacing slow global memory accesses with fast shared memory reads, the execution time of a kernel can be significantly reduced. To further increase the execution performance, these initial reads can also be coalesced. When global memory reads are coalesced, a

single memory transaction can return an entire section of memory for a warp as opposed to issuing multiple reads for each thread. Prior to the release of compute 2.0, coalescing global memory accesses required strict alignment of threads to consecutive memory addresses within a section of memory. After compute 2.0 these restrictions were relaxed and the penalties of not coalescing memory reads were mitigated by the inclusion of an additional L1 cache. To further reduce the number of memory reads issued, each thread within a warp should access the same 128 byte region of global memory (as illustrated in Fig. 2.5 and Fig. 2.6).

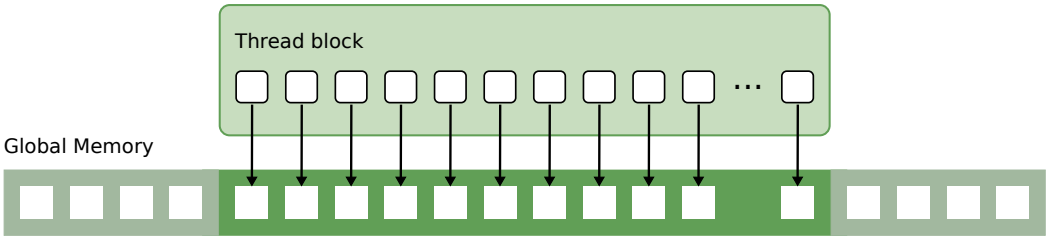


Figure 2.5: An example of all threads within a block accessing the same 128 byte region of global memory resulting in a single 128 byte load.

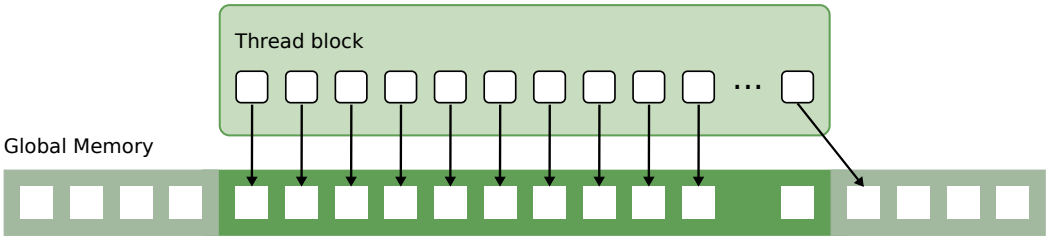


Figure 2.6: An example of the threads from the same block accessing two 128 byte regions of memory resulting in two separate global memory loads.

To summarise: when designing a kernel for parallel CUDA execution, it is important to fully and properly use the three main memory types (register, shared and global). Kirk and Hwu [5] note, global memory is the slowest of all the memory types but often large, whereas shared memory is fast but extremely limited and must be explicitly managed in order to maximise performance. Register memory is the fastest of all the memory types but has limited accessibility restricted to each thread, and exhausting the number of registers can result in costly spilling.

2.3.3 Synchronisation

Key to the design of any parallel algorithm for execution on the GPU is synchronisation. As there are multiple memory types and levels of abstraction (from CUDA threads to thread blocks), there are also multiple types of parallel synchronisation. In the following subsections we review each of the synchronisation types available.

Warp Synchronisation

Threads within a warp execute non-branching regions of kernels in parallel. There is implicit synchronisation between all threads as long as the warp execution path does not diverge. However, NVIDIA suggest avoiding warp-synchronous programming as this can lead to synchronisation issues and race conditions; furthermore, implementations often incorrectly assume that the number of threads in a warp will always be 32 [31]. NVIDIA note that although warp-synchronous implementations might function correctly now, changes to the CUDA toolchain and CUDA capable hardware might may easily break these implementations. However, warp synchronous programming can lead to improved execution times.

Thread Synchronisation

There is no guarantee as to what order warps will execute, necessitating the usage of additional block synchronisation techniques where inter-thread communication is required across multiple warps. The method *__syncthreads()* will block the execution of threads within a block up to a point in the kernel. After all threads have reached the *__syncthreads()* method the kernel execution will resume ensuring block synchronisation. The *__syncthreads()* method should not be called within an if-then-else statement unless all threads follow the same execution path as this may cause a deadlock resulting in the kernel timing out and failing to execute successfully. As of compute 2.0 there are three variants of the *__syncthreads()* method that allow similar voting functionality to warp vote. An example of a *__syncthreads()* method variant is *__syncthreads_count()*. This variant method takes an integer from each of the threads in the block, synchronises all threads to a point and returns the number of non-zero integers input to each of the threads.

Block synchronisation

There is no built-in support for block synchronisation in CUDA during kernel execution. This allows the GPU to execute each block independently and adjust the block execution scheduling dynamically according to the number of CUDA cores available on the GPU. As a result there is no guarantee as to the execution order of blocks or which blocks are currently executing at any given point and developers should not presume ordering. In Fig. 2.7 we illustrate a simple alternative to global synchronisation by splitting one kernel into two smaller kernels. A blocking synchronisation method *cudaDeviceSynchronize()* is used to ensure that all thread blocks have fully completed their kernel execution. A second kernel is then scheduled for execution by the host with the guarantee that the previous kernel had finished execution therefore achieving global synchronisation up to a point.

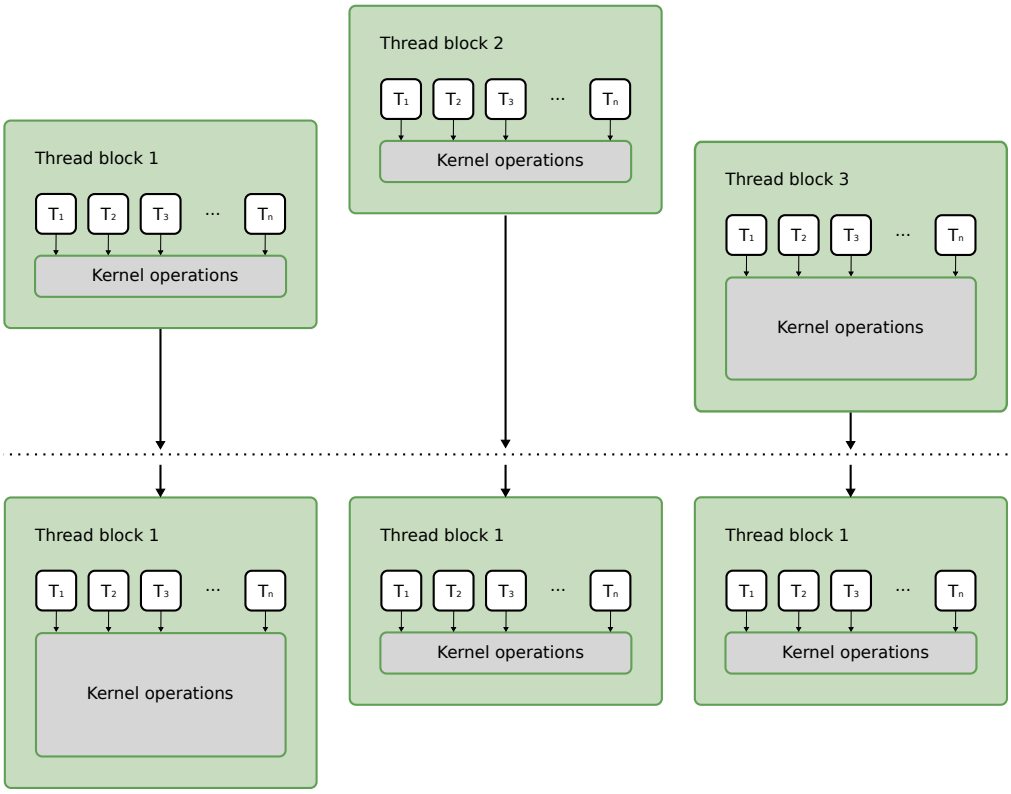


Figure 2.7: An example of global thread block synchronisation via executing the function cudaDeviceSynchronize() to explicitly synchronise all blocks.

One restriction with this method of achieving global synchronisation is that additional expensive memory operations may be required. As previously mentioned, the lifetime of shared memory is limited to the scope of a single kernel. Therefore the contents of shared memory will have to be copied to global memory in the first kernel and back from global memory to shared memory in the second kernel. To avoid this issue, other methods have been proposed to achieve global synchronisation such as using atomic operations, but such practices are discouraged by NVIDIA as they could be unsupported after subsequent new releases of CUDA.

Atomic Operations

As there is no guarantee as to which order threads, warps and thread blocks will execute, additional *atomic* operations are provided to read and update global memory values without interruption from other threads accessing the same region of memory. Atomic operations perform read-modify-write on a region of global memory. Each operation locks a region of memory until the initial operating thread has read the value, modified the value and written the value back to global memory [30]. After the lock is placed on a region of memory, all other threads must wait until this lock is removed before accessing the same region of memory [33].

A simple example of an atomic operator is *atomicAdd*. The *atomicAdd* operation reads a word in global memory, adds a value to this word and writes the value back to global memory (read-modify-write). Without atomic operations two threads could potentially read the same value simultaneously, independently modify the value and write back to global memory with the second write overwriting the value of the first write. In this scenario the value written by the first thread is lost. By using the *atomicAdd* method, the second thread would wait until the first thread has fully completed the read-modify-write operations ensuring the value written to global memory is the initial read value for the second thread. Atomic operations are more expensive than read and write global memory accesses. The use of atomic operations should be limited to the absolute minimum to avoid scenarios where multiple threads are waiting on a single region of memory. For example, instead of using *atomicMax*, threads within a block should first use shared memory to compute the maximum value so as to avoid each thread using an atomic operation.

2.3.4 OpenCL

After the release of CUDA, an alternative open standard general-purpose programming API was released under the name OpenCL (Open Computing Language) [5]. Initially Developed by Apple and subsequently the Khronos Group, OpenCL allows developers to harness the GPU and multi-core CPUs for general-purpose parallel computation. However unlike CUDA, OpenCL has multi-vendor and multiplatform support thus allowing parallel code to be executed on AMD and NVIDIA GPUs as well as x86 CPUs [5]. This gives OpenCL the advantage of portability between platforms. However as Kirk and Hwu [5] note, OpenCL programs can be inherently more complex if they choose to accommodate multiple platforms and vendors. Developers must use different features from each platform to maximise performance and so multiple execution paths dependent on the platform and vendor must be included. This can result in each platform achieving a different execution time depending on the input algorithm, mapping and usage of platform specific APIs that may give an advantage to that specific platform. Kirk and Hwu also note that the design of OpenCL is influenced heavily by that of CUDA and as a result working with OpenCL can be very similar to CUDA. As with CUDA, regions of the application that execute in parallel are encapsulated in kernels. OpenCL also has a similar concept of CUDA blocks and threads which have been renamed to *Work group* and *Work item* respectively. The current index of the block within the grid of all blocks has also been renamed as the *NDRange*. To facilitate support for multiple devices across platforms and vendors, OpenCL introduces the concept of an OpenCL context. Each device is assigned to a context and work is scheduled for execution in a queue for that context [5]. For additional information regarding OpenCL, we direct the reader to the Khronos Group OpenCL specification [34].

In this thesis we choose to use NVIDIA CUDA as the primary programming model along with NVIDIA GPUs. This is due to current API capabilities, better development and profiling tools and improved GPU hardware. This unfortunately comes with the sacrifice of portability between GPU vendors however as we previously noted the two parallel models are very similar. We envisage that our parallel implementations could be ported to use OpenCL and ATI GPUs if necessary.

2.4 Recent advances

In this section we detail the recent advances made to CUDA since 2007.

2.4.1 Fermi, Kepler and Maxwell

CUDA is now a proven technology in its sixth major toolkit release and as of September 2014, fourth major hardware revision (*Maxwell*). From the initial CUDA compatible GPU, each major hardware revision has made significant improvements to the speed, number of CUDA cores, on-chip cache sizes and clock speed of the GPU. Improvements have also been made to the programmability of the CUDA applications reducing the need for strict memory coalescing and providing better automatic caching of global memory accesses. In Table 2.1 we compare the high-end CUDA devices showing the considerable improvements from 2006 to 2014 [35].

	Tesla	*Fermi*	*Kepler*	*Maxwell*
Card Name	8800 GTX	GTX 580	GTX 680	GTX 980
Chip	G80	GF110	GK104	GM204
CUDA Cores	128	512	1536	2048
Base Clock	575 MHz	772 MHz	1006 MHz	1126 MHz
Memory Clock	900 MHz	2004 MHz	6008 MHz	7010 MHz
Shared Memory	16 KB	48 KB	48 KB	96 KB
Memory	768 MB	1536 MB	2048 MB	4096 MB
Memory Bandwidth	86.4 GB/s	192.4 GB/s	192.3 GB/s	224.3 MB

Table 2.1: A comparison of consumer level CUDA GPUs for each hardware revision.

Fermi

In an NVIDIA whitepaper, Patterson describes how the architectural hardware advancements from Tesla to the Fermi architecture are as significant as NV40 (pre-CUDA) to Tesla [36] and goes on to define the top innovations made in Fermi. Of these innovations significant improvements have been made in: the performance (and quality) of floating point operations; on-chip cache sizes; and the performance of atomic instructions. Fermi also introduced Error Correcting Codes (ECC) to

main and cached memory alongside a new instruction set for increased performance. The increase in floating performance results in double precision floats now running only half the speed of single. As previously mentioned Fermi increased the on-chip cache size from 16KB shared memory to 48KB of configurable L1 cache with an additional 768KB of L2 cache. By increasing the shared memory size, Patterson describes how applications with lower levels of parallelism can hide DRAM latency more efficiently. Additionally the L2 cache alleviates the need for complex shared memory caching patterns for applications with random access patterns albeit with a slight penalty for the L2 latency. By reducing the time to switch contexts, Fermi allows concurrent kernel execution with bi-directional parallel memory transfers. This benefits asynchronous applications where the results of one kernel are not necessarily used immediately, or in the case where memory can be transferred in small chunks, a kernel is invoked whilst the next transfer is being completed. The results from the kernel can then be transferred back to the host whilst the next chunk of memory is transferred across. Fermi also boasts atomic instructions which are around 5x to 20x faster than the Tesla as they can be stored in the L2 cache.

Kepler

Following on the success of the Fermi architecture, Kepler tripled the number of CUDA cores, increased GPU performance and decreased power consumption. NVIDIA note that the architecture of Kepler extends the same programming model as Fermi and so the best practices discussed for Fermi are applicable to Kepler. Kepler also introduced warp shuffle methods (for communication between threads in a warp), increased parallelism and increased shared memory bandwidth. The second generation of Kepler GK110 devices also introduced dynamic parallelism. Dynamic parallelism allows kernels to launch additional kernels and thus nest kernel execution. This allows the GPU to dynamically create additional work starting with a coarse grained first kernel and executing finer grained kernels for specific regions when required. A simple example would be processing an image with mostly empty regions: the first kernel would identify regions of interest, executing nested kernels when areas of interest are discovered. Kepler also allows up to 16 kernels to execute simultaneously so further increasing parallelism.

Maxwell

Maxwell is the most recent CUDA compatible GPU architecture initially announced by NVIDIA in mid-February 2014 with the GeForce 7 Series. A second more powerful iteration was released in mid-September 2014 along with the new GeForce 9 Series of GPU. As with Kepler, Harris notes [35] that developers will not have to change their existing CUDA implementations to benefit from the increased performance and efficiency benefits. However, to fully utilise the architecture developers should make use of the new hardware and software features available.

The Maxwell architecture brings significant CUDA hardware improvements. In Table 2.1 we show the increase in shared memory from the first generation of CUDA compatible GPUs to the latest iteration of Maxwell GPUs. The Maxwell architecture doubles the amount of shared memory available for each thread block. Applications that were previously exhausting the amount of shared memory will benefit from this change when built against the latest compute version, and Harris [35] notes that shared memory bound kernels can achieve up to double the occupancy compared to Kepler GPUs. Developers can no longer specify the split between L1 and shared memory and therefore shared memory is now locked at the higher amount of 96KB. In addition to increased shared memory, the L2 cache is now four times the size of the previous Kepler generation, so increasing performance of bandwidth-bound CUDA applications [35]. Another significant increase is the number of active thread blocks per SM to 32 (up from 16). As a result of this change, each SM will be able to execute double the number of thread blocks potentially improving occupancy of kernels with thread blocks with low thread counts [35].

Maxwell brings a new SM design from the previous Kepler generation and has been renamed as SMM (previously SMX). The new SMM design achieves around 1.4x performance per code compared to the older Kepler SMX. For a detailed comparison of the key differences between the SMM and SMX we refer the reader to the NVIDIA GTX 980 whitepaper [37]. Along with increased performance Maxwell also brings significantly increased power efficiency. Maxwell provides double the performance per watt against the Kepler generation of GPU [37] while maintaining the same 28-nm manufacturing process highlighting the efficiency of the new SM.

2.4.2 Embedded and low power applications

In mid-2013 NVIDIA announced [38] a new release of the CUDA Toolkit (version 5.5) with support for the ARM processor architecture alongside the existing x86 architecture support. This enabled parallel CUDA applications to be executed on a wider range of devices and form factors. ARM processors are small low-powered processors typically found in mobile devices that are based on the ARM processor architecture. The ARM processor architecture is similar to a reduced instruction set computing (RISC) architecture with various enhancements to allow the processors to archive higher performance and low power consumption [39]. ARM-based processors are manufactured based on the IP licensed directly from ARM and fabricated by partners such as Samsung, Qualcomm or NVIDIA. These processors are generally cheap and sold in large volumes. ARM report that around 16 million processors are sold every day with around 30 billion sold to date. Due to the low power consumption, efficiency, small size and cost, these low-powered processors are found in mobile devices such as phones and recently, smart watches.

Parallelism is also key to the efficiency of ARM processors with mobile devices now routinely using 2-8 CPU cores. ARM provide a SIMD engine known as NEON for parallel processing on ARM CPUs with emphasis on multimedia applications but also general purpose applications such as DSP processing, game processing and image processing [40]. Similarly to CUDA, by increasing the occupancy of the ARM processor using NEON, throughput can be increased and the power consumption of the CPU can be reduced. To further still increase performance and reduce the power consumption, ARM introduced a new architecture known as big.LITTLE which couples modern high performance ARM CPUs with lower power, smaller, efficient ARM CPUs switching between the two depending on the context [41].

ARM also produce their own GPUs known as Mali and can be included on the SoC (system on a chip). However, partners can chose to include their own GPUs (for example, NVIDIA use their own Tegra mobile GPU). Along with the initial ARM toolkit, release NVIDIA produced a small ARM-based device development board (codenamed Kayla) featuring an ARM CPU and an NVIDIA CUDA compatible GPU. Until this release, execution of CUDA code was limited to x86-based

devices which severely limited the potential execution of highly parallel CUDA applications to desktops and laptop devices. By extending support to multiple processor architectures (including ARM) CUDA can now target mobile, embedded and low power applications for potential speed increases. This greatly increased the scope of CUDA as optimised parallel code can now bring potential speed increases across a range of devices scaling to utilise the number of CUDA cores available.

Following the ARM announcement in 2014, NVIDIA released the Jetson TK1 development kit. Expanding on the previously released low powered ARM based board, the TK1 significantly improves the parallel capabilities and includes a new GPU with 192 CUDA cores based on the Kepler architecture.

2.4.3 CUDA roadmap

At NVIDIA GTC 2014, NVIDIA announced Pascall. The next generation GPU architecture and the successor to the recently released Maxwell architecture. Pascal is due to arrive in 2016 and brings a radically redesigned GPU layout featuring 3D memory and reducing the physical footprint of memory on the GPU. 3D memory stacks multiple wafers to increase bandwidth, capacity and energy efficiency. As a result of the redesigned GPU layout, Pascal GPUs are around one third of the size of an existing PCIe card. Pascal will also introduce NVLink, a high speed alternative to PCIe with significantly increased bandwidth for sharing data from the GPU to GPU and GPU to CPU (providing that the CPU is NVLink compliant).

CHAPTER 3

Parallel Ant Colony Optimization on the GPU

This chapter is based on the following publication:
Improving Ant Colony Optimization performance on the GPU using CUDA [1].

3.1 Ant Colony Optimisation

Ant algorithms model the observed behaviour of real ants to solve a wide variety of optimisation and distributed control problems. Dorigo and Stützle [19] note that ant algorithms replicate the self-organizing and co-ordinated principles of real ant colonies. This allows artificial ants to solve complex tasks where good solutions are an emergent property of these basic principles. An ant algorithm consists of two main stages; building a solution and feeding back information to other ants. This process is repeated and the next iteration of solution construction is influenced by the information shared from the last iteration. This feedback mechanism helps ants to improve their solutions constructed. As a result the ants will communicate improved solutions over time and converge on better solutions. Ant Colony Optimisation (ACO) [42] is a population-based metaheuristic that can be used for modelling discrete optimisation problems. One of the many problems ACO has been successfully applied to is NP-hard the Travelling Salesman Problem (TSP) in which the goal is to find the shortest tour around a set of cities thus minimising the total distance travelled. Dorigo and Stützle [19] remark that when modelling new algorithmic ideas, the TSP is the standard problem to model due its simplicity; often algorithms that perform well when modelling the TSP will translate with similar success to other optimisation problems. When modelling the TSP, the simplest implementation of ACO, Ant System (AS), consists of two main stages: *tour construction* and *pheromone*

update (an optional additional local search stage may also be applied once the tours have been constructed so as to attempt to improve the quality of the tours before performing the pheromone update stage). The process of tour construction and pheromone update is applied iteratively until a termination condition is met. There are many different termination conditions including; performing a set number of iterations of the algorithm, terminating when a solution with a set range is constructed or when an optimal solution is obtained. By a process known as *stigmergy*, ants are able to communicate indirectly through a *pheromone matrix*. This matrix is updated once each ant has constructed a new tour and will influence successive iterations of the algorithm.

3.2 Solving the Travelling Salesman Problem

In order to solve the TSP we aim to find the shortest tour around a set of cities from a given start point and returning to the starting point. A complete tour contains each of the available cities. Each city must be visited once (excluding the starting city) to produce a tour. An instance of the problem is a set of cities where for each city we are given the distances from that city to every other city. These inter-city distances are often represented using a matrix of size $N \times N$. In more detail, an instance is a set N of cities with an edge (i, j) joining every pair of cities i and j so that this edge is labelled with the distance $d_{i,j}$ between city i and city j. Whilst the aim is to solve the TSP by finding the shortest-length Hamiltonian circuit of the graph (where the length of the circuit is the sum of the weights labelling the edges involved), the fact that solving the TSP is NP-hard means that in practice we can only strive for as good a solution as possible within a feasible amount of time (there is usually a trade-off between these two parameters). Throughout our research we only ever consider *symmetric* instances of the TSP where $d_{i,j} = d_{j,i}$, for every edge (i, j), as opposed to non-symmetric instances where $d_{i,j}$ does not equal $d_{j,i}$. The AS algorithm came after three initially-proposed ant algorithms [19] ant-density, ant-quantity and ant-cycle. With ant-density and ant-quantity the pheromone levels were updated after every ant move. Ant-cycle improved on this and only updated the pheromone levels after all ants have moved. The amount of pheromone deposited by each ant in ant-cycle was proportional to the quality of the tour further increasing the quality of the tours constructed. AS now refers to ant-cycle due to the convergence performance and solution quality over the other two variants. AS and consists of two main stages (see Fig. 3.1): *ant solution construction*; and *pheromone update*.

```
procedure ACOMetaheuristic
  set parameters, initialise pheromone levels
  while (termination condition not met) do
    construct ants' solutions
    update pheromones
    local search (optional)
   end
end
```

Figure 3.1: An overview of the Ant System algorithm. [19]

Solution construction and pheromone update are iteratively repeated until a final termination condition is met such as a minimum tour length or number of iterations. The solutions can be optionally improved after each iteration by performing a local search operation such as 2-opt, 3-opt or k-opt. The 2-opt search algorithm removes two edges in the final tour and reorganises the edges to potentially reduce the total tour length (with 3-opt removing 3 edges and so on). For example, by removing the edge (c, d) and (h, g) we can insert two new edges between (c, g) and (d, h) (illustrated in Fig. 3.2). If the length of the new tour (with the two new edges inserted) is lower than the tour prior to performing 2-opt (with the original two edges) the changes are kept. If the length of the tour is not lower, the changes can be disgarded and the process repeated until some termination condition is met. The colony contains m artificial ants, where m is a user-defined parameter. In the tour construction phase, these m ants construct solutions independently of each other.

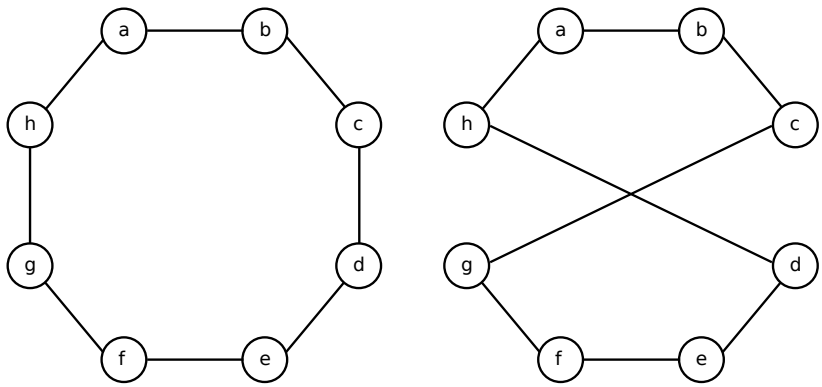


Figure 3.2: An example of performing the 2-opt operation on a tour. The edges between (c, d) and (h, g) and are replaced with new edges between (c, g) and (d, h).

3.2.1 Solution Construction

To begin, each ant is placed on a randomly selected initial starting city. All ants within the colony then repeatedly apply the random proportional rule independently, which gives the probability of ant k moving from its current city i to some other city j, in order to construct a tour (the next city to visit is chosen by ant k according to certain probabilities). Each ant in the colony will perform this action $n-1$ times (where n is the number of cities) before returning to the respective initial randomly selected start city thus constituting a complete and valid tour. At any point in the tour construction, ant k will already have visited some cities. A complete tour is considered valid if each city is visited exactly once (excluding the starting city). To stop an ant re-visiting previously visited cities, we maintain a set of legitimate cities to which an ant may next visit and this is denoted as N^k. This list is updated after the next city is selected to avoid repeatedly re-visiting the same city. Suppose that at some point in time, ant k is at city i and the set of legitimate cities is N^k. The *random proportional rule* for ant k moving from city i to some city $j \in N^k$ is defined via the probability:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \tag{3.1}$$

where: τ_{il} is the amount of pheromone currently deposited on the edge from city i to city l; η_{il} is a parameter relating to the distance from city i to city l and which is usually set at $1/d_{il}$; and α and β are user-defined configurable parameters to control the influence of τ_{il} and η_{il}, respectively. Dorigo and Stützle [19] suggest the following parameters when using AS: $\alpha = 1$; $2 \leq \beta \leq 5$; and $m = |N|$ (that is, the number of cities), i.e., one ant for each city and thus increasing the number of tours per iteration as the size of the TSP instance increases. The probability p_{ij}^k is such that edges with a smaller distance value are favoured and thus have a greater chance of being the next city selected by ant k.

3.2.2 Pheromone Update

For the second stage of the Ant System algorithm, the results of solution construction must be saved to the pheromone matrix to influence subsequent iterations of tour construction as ants cannot directly communicate with each other. Once all of the ants have constructed their tours, the pheromone levels of edges must be updated. To avoid stagnation of the population, the pheromone level of every edge is first evaporated

according to the user-defined *evaporation rate* ρ (which, as advised by Dorigo and Stützle [19], we take as 0.5). So, each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij}. \tag{3.2}$$

Over time, this allows edges that are seldom selected to be forgotten. Once all edges have had their pheromone levels evaporated, each ant k deposits an amount of pheromone on the edges of their particular tour T^k so that each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k, \tag{3.3}$$

where the amount of pheromone ant k deposits, that is, $\Delta\tau_{ij}^k$, is defined as:

$$\Delta\tau_{ij}^k = \begin{cases} 1/C^k, & \text{if edge } (i,j) \text{ belongs to } T^k \\ 0, & \text{otherwise,} \end{cases} \tag{3.4}$$

where C^k is the length of ant k's tour T^k. The method of updating the pheromone levels ensures that a shorter tour found by some ant will result in a larger quantity of pheromone being deposited on the edges traversed in this tour. This in turn will increase the chances of one of these edges being selected by some ant (in the next iteration) according to the random proportional rule and becoming an edge in some subsequent tour. Dorigo and Stützle (see [19]) note that for larger instances of the TSP, the performance of AS decreases and thus reduces the viability of the algorithm. To constitute a valid and complete tour, each ant must visit all of the cities available exactly once and so the workload for each ant per iteration will increase as the tour size increases. As previously mentioned, the number of ants recommended is equal to the size of the TSP instance increasing the computational load as the TSP instance increases in size. Many alternative algorithms, techniques and extensions have subsequently been proposed to reduce the impact on performance including *local search*, *elitist* AS, *rank-based* AS, $\mathcal{MAX} - \mathcal{MIN}$ AS ($\mathcal{MAX} - \mathcal{MIN}$), *ant colony system* and the *hyper-cube framework for ACO* [19].

3.3 Parallel Ant Colony Optimisation

When sequentially executing AS, each ant in the colony iteratively builds up its own tour with each ant being executed consecutively [19]. It is not until all ants have completed building their tours that pheromone is deposited. Therefore each ant cannot influence other ants during the current iteration of solution construction or pheromone update and cab be executed independently sequentially or in parallel. This makes each stage of ACO particularly suited to parallelisation. Dorigo and Stützle [19] note that there are two main approaches to implementing ACO in parallel: the first maps each ant to an individual processing element (a collection of processing elements therefore constitutes a single colony of ants); the second maps an entire colony of ants to a processing element (usually augmented with a method of communicating between the colonies). Multiple colonies are often executed in parallel, potentially reducing the number of iterations before termination as each colony works towards a solution.

3.3.1 Parallel ACO on the GPU

In this chapter we detail our work on improving parallel ACO implementations for solving the TSP on the GPU using NVIDIA CUDA. Recent works of Cecilia et al. [20] and Delévacq et al. [21] have shown how ACO can be successfully implemented using a data-parallel approach (see Section 3.4.2) on the GPU using NVIDIA CUDA. The novel data-parallel approach maps individual ants to CUDA thread blocks resulting in significant speedups over existing methods which map individual ants to CUDA threads.

Motivated by these findings we present an improved data-parallel approach executing both the tour construction (see Section 3.2.1) and pheromone update (see Section 3.2.2) stages of the Ant System algorithm in parallel on the GPU for solving the TSP.

For the tour construction phase, our approach uses a new parallel implementation of the commonly-used roulette wheel selection algorithm (proportionate selection) called *DS-Roulette*. DS-Roulette exploits the modern GPU hardware architecture, increases parallelism, and decreases the execution time whilst still enabling us to construct high-quality tours. For the pheromone update phase, we adopt the approach taken by $\mathcal{MAX} - \mathcal{MIN}$ Ant System, and allied with a novel implementation we achieve significant speed-ups against a sequential counterpart and existing parallel implementations on the GPU.

3.4 Related work

As regards parallel implementation, ACO algorithms can be categorised as *fine grained* or *coarse grained*. In a fine grained approach, the ants are individually mapped to processing elements with communication between processing elements being ant to ant. In a coarse grained approach, entire colonies are mapped to processing elements with communication between colony to colony [19] (adopting a coarse grained approach with no communication between colonies is equivalent to executing the algorithm multiple times on the same instance but independently). In this section we will review existing parallel ACO contributions that target the GPU in more detail.

Catala et al. [43] presented one of the first GPU implementations of ACO targeted at modelling the *Orienteering Problem*. At the time of publication, the unified GPU was not available and CUDA had not yet been released. The implementation presented relies upon a previous direct GPU interface using graphics paradigms to model and solve general-purpose problems. Jiening et al. [44] implemented the MMAS algorithm to solve the TSP. This early work was also published prior to the release of CUDA and the authors note that their implementation was much more complex than its CPU counterpart. Their implementation of a parallel tour construction phase resulted in a small but incremental speedup. Using the *Jacket* toolbox for *MATLAB*, Fu et al. [45] implemented a parallel version of MMAS for the GPU to solve the TSP. Their approach focused more on a MATLAB implementation and less on the GPU (as parallelisation is handled by Jacket). They reported a speedup; however, their comparative CPU implementation was MATLAB-based which is inherently slower due to being an interpreted language.

Zhu and Curry [46] modelled an ACO Pattern Search algorithm for nonlinear function optimisation problems using CUDA on the GPU. They reported speedups of around 250x over the sequential implementation. Bai et al. [47] implemented a coarse-grained multiple colony version of MMAS using CUDA to solve the TSP. Each ant colony is mapped to a thread block and within the block each thread is mapped to an ant. Their approach yields tours with a quality comparable to the CPU implementation but the speedup reported is only around 2x. You [48] presented an implementation of the AS algorithm using CUDA to solve the TSP. Each thread is mapped to an ant but each thread block is part of a larger colony. You [48] notes that the use of shared memory is important for frequently accessed information by each ant such as visited cities. The speedup reported for this approach is around 2-20x when the number of ants is equal to the number of cities.

Weiss [49] developed a parallel version of AntMinerGPU (an extension of the MMAS algorithm). As in [47] and [48], each ant within the colony is mapped to an individual CUDA thread. Weiss argues that this approach, coupled with the AntMinerGPU algorithm, allows for larger population sizes to be considered. Weiss moves all stages of the algorithm to the GPU to avoid costly transfers to and from the GPU, a practice that is advocated by the programming guide to speed up CUDA applications [30]. Weiss tested the implementation against two problems from the UCI Machine Learning Repository: Wisconsin Breast Cancer; and Tic-Tac-Toe. The GPU version on AntMiner was up to 100x faster than the CPU implementation on large population sizes. Finally, Weiss noted that the implementation could easily be extended to support multiple colonies across multiple GPUs with possible inter-GPU communication.

3.4.1 Large TSP instances

O'Neil et al. [50] present a highly parallel hill climbing algorithm to solve the TSP using CUDA. They were able to attain speedups of up to 62x over optimised sequential code and note that it takes 32 CPUs each with 8 processing cores (256 cores in total) to match the performance of their parallel CUDA implementation. Their implementation also features a highly parallel implementation of the 2-opt algorithm which is able to process up to 20 billion 2-opt moves per second on a single CUDA compatible GPU.

Rocki and Suda [51] extend the work of O'Neil et al. [50] and also present an iterative hill climbing algorithm on the GPU using CUDA but expand the maximum number of cities from 110 to 6000 cities. Rocki and Suda [51] note that in order to process larger city instances it was critical to move the city distances out of shared memory and onto slower global memory. However in doing so this introduced additional latency as the global memory read times are significantly slower than shared memory.

For TSP instances with many cities O'Neil et al. [50] note that an iterative hill climbing approach may be better suited to the GPU due to the speed of their implementation against other approaches such as genetic algorithms or ant colony optimization approaches. However at the time of publication there were only task-based parallel approaches for implementing ACO on the GPU as Cecilia et al. note [20] and Delèvacq et al. [21] had yet to publish their findings on data-parallelism (see Section 3.4.2) which was able to significantly reduce the execution time of running ACO in parallel on the GPU by mapping individual ants to thread blocks as opposed to CUDA threads.

3.4.2 Data-parallelism

As Cecilia et al. note [20], the majority of existing contributions fail to implement the entire ACO algorithm on the GPU and provide no systematic analysis of how best to implement the algorithms in parallel. All of the above implementations also adopt a task-based parallelism strategy that maps ants directly to threads. The following two papers adopted a novel data parallel approach that maps ants to thread blocks so as to better utilise the massively parallel GPU architecture (as discussed in Chapter 2). By using this approach the total number of ants is equal to the number of thread blocks as opposed to the number of thread blocks multiplied by the number of threads per block.

Cecilia et al. [20] implemented the AS algorithm for solving the TSP on the GPU using CUDA. They note that the existing task-based approach of mapping one ant per thread is fundamentally not suited to the GPU. With a task-based approach, each thread must store each ant's memory (e.g., list of visited cities). This approach works for small tours but quickly becomes problematic with larger tours, as there is limited shared memory available and exhausting the shared memory can result in costly register spilling. The alternatives are to use fewer threads per block, which reduces GPU occupancy, or to use global memory to store each ant's memory, which dramatically reduces the kernel's performance as global memory is significantly slower than efficiently using shared memory.

The second issue with task-based parallelism (as discussed in [20]) is warp-branching. As ants construct a tour, their execution paths generally differ due to conditional statements inherent when using roulette wheel selection on the output of the random proportional rule (Fig. 3.1). When a warp branches, all threads within the branch are serialised and executed sequentially until the branching section is complete, thus significantly impeding the performance of branching code due to the loss of parallelism. Cecilia et al. present data-parallelism as an alternative to task-based parallelism so as to avoid these issues and potentially increase performance by better utilising the GPU architecture.

Data parallelism avoids warp-divergence and memory issues by mapping each ant to a thread block. All threads within the thread block then work in cooperation to perform a common task such as tour construction. Cecilia et al. use a fixed-size thread block that tiles to match the size of the problem. Tiling is a commonly used CUDA technique that allows a fixed size set of threads to repeat over a larger size input data set so that the input size is not restricted by the total thread count (where each thread performs some action on the input data in parallel). For example, when using an data input set size of 1024 and a fixed size thread block of 256 threads; each of the threads would tile across

the input data four times. In this case, thread 0 would process index 0, 256, 512 and 768. This technique is useful as the number of threads is limited by the version of CUDA. A thread is responsible for an individual city and the probability of visiting a city can be calculated without branching the warp by using a new proportionate selection method known as I-Roulette [20]. Cecilia et al. also implement the pheromone update stage on the GPU thus ensuring that both stages of AS execute on the GPU to avoid costly memory transfers. Two alternative pheromone update implementations are presented but Cecilia et al. conclude that the much simpler method of using atomic methods is significantly faster and will likely to continue to be so due to ongoing hardware improvements to atomic instructions by NVIDIA with newer GPUs. A speedup factor of up to 20x is reported when both the tour construction and pheromone update phases are executed on the GPU. The majority of the execution time reported by Cecilia et al. was spent on the tour construction phase.

The second data-parallel paper was presented by Delèvacq et al. [21] and implemented the $\mathcal{MAX} - \mathcal{MIN}$ algorithm with 3-opt local search for solving the TSP using CUDA with improved solution quality. Delevacq et al. conduct a similar survey of data-parallelism against task-based parallelism and also conclude that a data-parallelism approach is more suitable, due to the aforementioned reasoning and include detailed benchmarks to support this assertion. As a result, no solutions are proposed using the task-based parallel mapping. To improve solution quality in [21], after a block constructs a new ant tour, 3-opt local search is applied. As Delèvacq et al. [21] note that the 3-opt search algorithm is a time consuming process but can yield near optimal results. Delèvacq et al. outline that the data structures required to compute the expensive 3-opt search must be stored in global memory, therefore the execution is slow.

Building on their initial offering, Delv̀acq et al. also implemented multiple colonies to run in parallel across separate GPUs. The same test instances of the TSP were considered (as were considered for the implementations in [20] mentioned above) and the results showed up to a 20x speedup against a sequential implementation of the MMAS algorithm with local search disabled. With 3-opt local search enabled, the quality of the tours constructed improved significantly; however, the speedup was notably reduced.

3.5 Implementation

In this section we present our parallel implementation of the AS algorithm for execution on the GPU. We base our parallelisation strategy on the works of Cecilia et al. [20] and Delvacq et al. [21] and adopt a data-parallel approach for the tour construction phase, mapping each ant in the colony to a thread block. As discussed in Section 3.4.2, we execute each stage of the algorithm (see Fig. 3.1) on the GPU to maximise performance.

The first stage of the algorithm parses the city data and allocates memory and the relevant data structures. For any given city set of size m, the city-to-city distances are loaded into an $n \times n$ matrix. As mentioned in Section 3.2 we are using symmetric instances of the TSP and so $d_{i,j} = d_{j,i}$, for every pair of distinct cities i and j. As the amount of global memory available exceeds the largest tested instances of the TSP we are able to use a matrix. However, for even larger test instances, alternative data structures should be considered that only store the value of $d_{i,j}$ and not $d_{j,i}$ (as the two values are equal and storing both leads to value duplication unnecessarily consuming memory).

Ant memory is allocated to store each ant's current tour and tour length. A *pheromone matrix* is initialised on the GPU to store pheromone levels and a secondary structure called *choice_info* is used to store the product of the denominator of equation 3.1 (an established optimisation, detailed in [19], as these values do not change during each iteration). Once the initialisation is complete, the pheromone matrix is artificially seeded with a tour generated using a greedy search as recommended in [19] (other approaches can also be used such as seeding large values to encourage search space exploration).

3.5.1 Tour construction

In Section. 3.2 we gave a broad overview of the tour construction phase that is applied iteratively until a new tour is constructed. Dorigo and Stützle provide a detailed description of how to implement tour construction sequentially along with pseudo-code [19] which we summarise in Fig. 3.3 (where the number of cities is n with the cities named as $1, 2, \ldots, n$). After the initial random city is chosen, the first inner for-loop repeats $n - 2$ times to build a complete tour (note that there are only $n - 2$ choices to make as once $n - 2$ cities have been chosen there is no choice as regards the last as the ant must return to the original city the tour started at to constitute a valid tour). Within the inner for-loop, the probability of moving from the last visited city to all other possible cities is calculated. Calculating the probability consists of two stages: retrieving the value of

choice_info[j][l] (the numerator in equation 3.1) and checking if city l has already been visited in the current iteration (in which case the probability is set to 0). The next city to visit is selected using roulette wheel selection.

```
procedure ConstructSolutions
  tour[1] ← place the ant on a random initial city
  for j = 2 to n − 1 do
    for l = 1 to n do
      probability[l] ← CalcProb(tour[1 . . . j − 1],l)
    end-for
    tour[j] ← RouletteWheelSelection(probability)
  end-for
  tour[n] ← remaining city
  tour_cost ← CalcTourCost(tour)
end
```

Figure 3.3: Overview of an ant's tour construction.

Roulette wheel selection is a prime example of something that is trivial to implement sequentially yet requires additional consideration when implemented in parallel using a GPU. As the operation is performed $n-2$ times for each ant per iteration, it is essential to find an efficient parallel implementation that executes fully on the GPU. *Roulette wheel selection* is illustrated in Table 3.1 where 1 item has to be chosen from 5 in proportion to the value in the first column labelled 'input'. The first step is to reduce the set of input values so as to obtain cumulative totals (as is depicted in the second column labelled 'reduced'). The reduced values are normalised so that the sum of all input values normalises to 1 (see the third column labelled 'normalised') and the portion of the roulette wheel corresponding to some item is calculated (see the fourth column labelled 'range'). The final step is to generate a random number that is greater than 0.0 and at most 1.0, and then to use the ranges so as to choose an item (so if 0.5 was generated, for example, then we would choose item 5, as 0.5 is greater than 0.365 but less than or equal to 0.875). Items with larger input values will have a larger range compared to smaller input values and an increased chance of being selected when a random number is generated. Critically this process allows poor values to be randomly selected. Sequentially this process is trivial; however, the linear nature of the algorithm, the divergence in control flow, parallel random number generation and the need for constant thread synchronisation means we have to work harder in our GPU setting.

Input	Reduced	Normalised	Reduced range
0.1	0.1	0.1	> 0.0 & ≤ 0.1
0.3	0.4	0.25	> 0.1 & ≤ 0.25
0.2	0.6	0.375	> 0.25 & ≤ 0.375
0.8	1.4	0.875	> 0.375 & ≤ 0.875
0.2	1.6	1.00	> 0.875 & ≤ 1.0

Table 3.1: An example run of roulette wheel selection (proportionate selection).

Cecilia et al. [20] address these issues by implementing a parallel implementation of roulette wheel selection, *Independent-Roulette* (I-Roulette) specifically for the GPU using CUDA. In I-Roulette, each thread l calculates the probability of moving to city j. Each thread then checks to see whether the city j has been previously visited and multiplies the probability just calculated by a random number. Finally, the n resulting values undergo a reduction and the largest value is selected as the next city (by a reduction we mean a process by which the values are pairwise linearly compared with the largest being retained). However, there are some issues with this approach.

- The reduction is performed on the entire thread block and this prohibits synchronisation between warps so that idle warps result.
- The random numbers generated can dramatically decrease the influence of the heuristic and the pheromone information. This in turns leads to a reduction in the quality of the tours generated.
- A random number must be generated for each thread for $n - 2$ iterations in order to build a complete tour.
- From the source code provided by Cecilia et al., it can be observed that often a single thread is used for various stages of the algorithm resulting in warp divergence and reduced parallelism. Where possible all threads within a warp should be used to achieve maximum performance.
- The entire tour is stored in shared memory. Larger tours that exceed the limited total amount of shared memory will spill out of shared memory resulting in a loss of performance over smaller tours.
- When constructing tours for large instances, shared memory is often exhausted.

3.5.2 Double-Spin Roulette

To address these issues we present *Double-Spin Roulette* (DS-Roulette), a highly parallel roulette selection algorithm that exploits warp-level parallelism, reduces shared memory dependencies, and reduces the overall instruction count and work performed by the GPU. In the sequential implementation of roulette wheel selection, each ant constructs a tour one city at a time and each ant is processed consecutively. The first level of parallelism we employ is to execute the tour construction stage in parallel. Using a data-parallel approach (as suggested in [20] and [21]), each ant is mapped to a thread block (so that m ants occupy m blocks) as opposed to executing multiple ants in a single thread block.

DS-Roulette consists of three main stages that are executed in succession (see Fig. 3.4). Thread synchronisation is utilised after each stage to ensure that all threads within the block have finished the previous stage before proceeding. As warps can execute in any order we cannot assume they will arrive at a certain point together thus necessitating an implicit synchronisation of all threads via the *__syncthreads()* operation. Cecilia et al. note that due to having a fixed maximum thread count and limited shared memory available, tiling threads across the block to match the number of cities yielded the best results. Without employing this tiling method the maximum number of cities per tour would be limited to the number of threads in a thread block specified before launch. Building upon this observation, the first stage of DS-Roulette tiles 4 thread warps (128 threads) so as to provide complete coverage of all potential cities (again illustrated in Fig. 3.4). We will henceforth refer to a tiled warp consisting of 32 threads as a sub-block that represents a block of possible cities.

In the first stage, each thread within the sub-block checks if the city it represents has previously been visited in the current tour. This valued is stored in shared memory and is known as the tabu value. Tabu values are stored in shared memory as the values are accessed frequently. A warp-level poll is then executed to determine if any valid cities remain in the sub-block (this reduces redundant memory accesses and execution time). If valid cities remain then each thread retrieves the respective probability from the *choice_info* array and multiplies the probability by the associated *tabu value*. The sub-block then performs a warp-reduction (see Fig. 3.5) on the probabilities to calculate the sub-block probability and this is stored in shared memory. Warp-level parallelism implicitly guarantees that all threads execute without divergence throughout the stage and this significantly decreases the execution time. Once the probability for the sub-block is calculated, the sub-blocks then tile to ensure complete coverage of all cities.

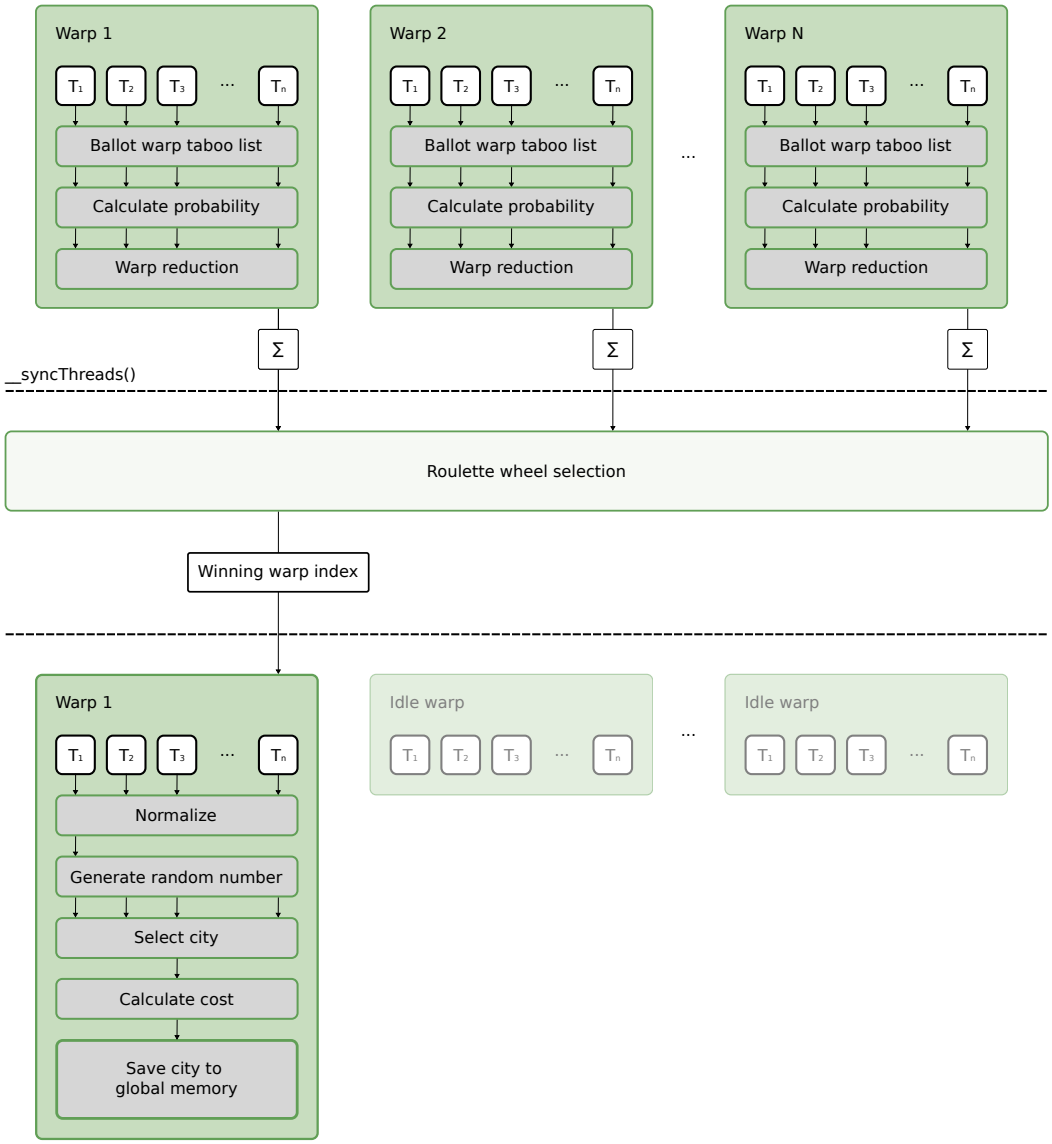


Figure 3.4: An overview of the double-spin roulette algorithm executed by each ant (thread block) during each iteration of the tour construction phase of the AS algorithm in order to construct a valid tour.

```
__device__ void
warp_reduce (int tid, float in, float *data){
  int idx = (2 * tid - (tid & 31) );
  data[idx] = 0;
  idx += 32;
  float t = data[idx] = in;

  data[idx] = t = t + data[idx - 1 ];
  data[idx] = t = t + data[idx - 2 ];
  data[idx] = t = t + data[idx - 4 ];
  data[idx] = t = t + data[idx - 8 ];
  data[idx] = t = t + data[idx - 16 ];
}
```

Figure 3.5: The warp-reduce method [52].

At the end of stage one, the results are a list of sub-block probabilities. Stage 2 performs roulette-wheel selection on this set of probabilities to chose a specific sub-block from which the next city to be visited will be chosen. If the tour size were 256 cities, for example, there would be 8 potential sub-blocks previously calculated by 4 warps. This is the first spin of the roulette wheel (other methods such as greedy selection can also be utilised here). Once a warp has been chosen, the value is then saved to shared memory.

In the final stage of DS-Roulette, we limit the execution to the chosen sub-block. Using the block value calculated in stage 2, the first 32 threads load, from shared memory, the probabilities calculated by the winning sub-block in stage 1. Each thread then loads the total probability of the sub-block and the probabilities are normalised. As each thread is accessing the same 32-bit value from shared memory, the value is broadcasted using a single memory read to each thread eliminating bank conflicts and serialisation. A single random number is then generated and each thread checks if the number is within its range thus completing the second spin of the roulette wheel. The winning thread then saves the next city to shared memory and global memory, and updates the total tour cost. After $n-2$ iterations, the tour cost is saved to global memory using an atomic max operator. This value is subsequently used during the pheromone update stage.

Using this technique we address the issues raised by I-Roulette. Instead of reducing and normalising the entire thread block, we split the block into smaller sub-blocks from which we approximate where the next city is to be selected from. Roulette wheel selection is applied twice to calculate the next city. This process drastically reduces

the synchronisation overhead and the need for each thread to generate a costly random number. We exploit warp-level scheduling to avoid additional computation and thus reduce the total instruction count. DS-Roulette preserves the influence of both the heuristic and the pheromone information and leads to higher quality tours. At any point, only the last visited city is required to be kept in shared memory thus reducing the total shared memory required. This in turn increases the occupancy of the GPU when larger tour instances are used.

3.5.3 Pheromone update

The last stage of the AS algorithm is pheromone update which consists of two stages: pheromone evaporation; and pheromone deposit. The pheromone evaporation stage (see equation (3.3)) is trivial to parallelise as all edges are evaporated by a constant factor ρ. A single thread block is launched which maps each thread to an edge and reduces the value using ρ. A tiling strategy is once again used to cover all edges (an alternative strategy was originally used to map each row of the pheromone matrix to a tiling thread block; however, this was found to be considerably slower in practice).

The second stage, pheromone deposit (see equation (3.4)), deposits a quantity of pheromone for each edge belonging to a constructed tour for each ant. Cecilia et al. [20] note that as each ant will perform this stage in parallel, atomics must be used to ensure correctness of the pheromone matrix. Atomic operations are expensive and Cecilia et al. provide an alternative approach using *scatter to gather transformations*. Using this approach removes the dependency on atomic operations; however, it results in more global memory loads which impedes the performance. As a result they conclude that although there is a dependency on atomics, the implementation is faster than other alternatives.

Other ant algorithms such as elitist ant, MMAS and ACO apply pre-conditions on the pheromone update state. For example, when using MMAS only the iteration's best ant deposits pheromone. To reduce the usage of atomic operations and increase convergence speed, we adopt the pheromone update stage from MMAS into our AS implementation. At the end of the tour construction stage, each ant performs a single atomic *min* operation on a memory value storing the tour length. This single operation per block allows the lowest tour value to be saved without additional kernels and is inexpensive. For the tour construction stage we then launch m thread blocks (representing m ants) which then individually check if their tour cost is equal to the lowest overall cost and if so deposit pheromone without the need for atomics as only one ant will be depositing pheromone.

3.6 Results

In this section, we present the results obtained by executing our implementation on various instances of the TSP and we compare these results to other parallel and sequential implementations. We use the previously mentioned parameters (see Section 3.2.1) but modify the value of ρ from 0.5 to 0.1 to reduce the rate of evaporation on the pheromone matrix (reducing the rate of evaporation will preserve edges within the pheromone matrix for longer) for both GPU and CPU implementations. As we have modified the pheromone deposit stage to only use the best ant, less pheromone is deposited. The reduced evaporation rate reflects this change and ensures that the pheromone matrix still has sufficient pheromone to influence the tour construction. We arrived at this value experimentally and found it to produce results comparable to the sequential implementation..

3.6.1 Experimental Setup

For testing our implementation we use an NVIDIA GTX 580 GPU (Fermi, see Fig 2.1) and an Intel i7 950 CPU (Bloomfield). The GPU contains 580 CUDA cores and has a processor speed of 1544 MHz. As the card is from the Fermi generation, it uses 32 threads per warp and up to 1024 threads per thread block with a maximum shared memory size of 64 Kb. The CPU has 4 cores which support up to 8 threads with a clock speed of 3.06 GHz. Our implementation was written and compiled using the latest CUDA toolkit (v5.0) for C and executed under Ubuntu 12.10. To match the setup of Cecilia et al. [20], timing results are averaged over 100 iterations of the algorithm with 10 independent runs.

3.6.2 Solution Quality

To evaluate the quality of the tours, we compared the results of our new GPU implementation against an existing CPU implementation of AS for the set number of iterations (where an iteration is a single run of solution construction and pheromone deposit for each ant). Our new approach was able to match and in most cases reduce the length of the tours when using identical parameters and number of iterations. As is expected when using the AS algorithm, the results are not optimal and the quality of tours constructed decreases as the number of cities increases. However, this was seen across both versions and can be improved by implementing local search. Fig3.6 shows a comparison of the average quality of tours obtained via the existing CPU and new GPU implementation.

Instance	Optimal	CPU	GPU
$d198$	15780	17583	16222
$a280$	2579	3015	2710
$lin318$	42029	46651	44495
$pcb442$	50778	62255	56639
$rat783$	8806	10896	9390
$pr1002$	259045	333262	341080
$pr2392$	378032	511977	537127

Table 3.2: The best tour lengths obtained for various test instances of the TSP on the CPU and GPU implementations over 10 independent runs with 100 iterations.

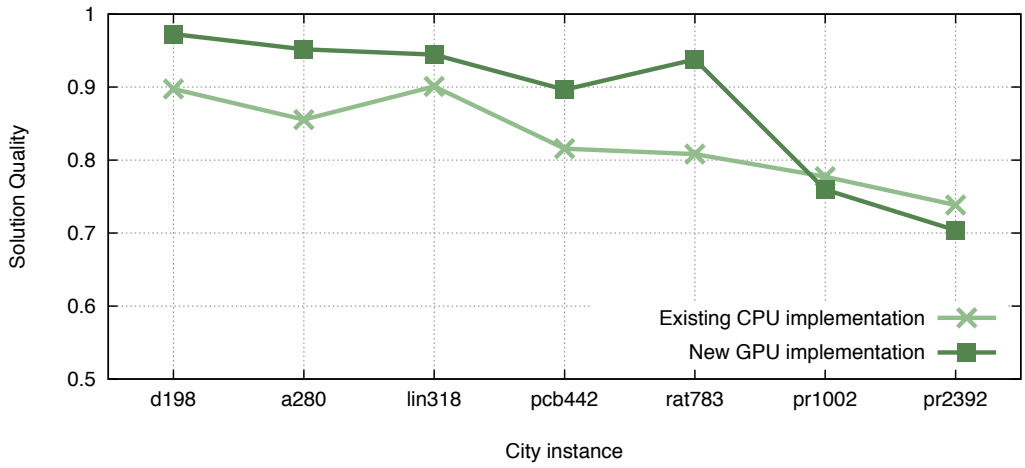


Figure 3.6: A comparison of the quality of tours constructed via the existing CPU implementation of AS and new GPU implementation.

3.6.3 Benchmarks

In Table 3.3 we present the execution times obtained when executing instances of the TSP using the setup documented in Section 3.6.1. In Fig 3.7 and Fig 3.8 we present the speedup attained against the standard CPU implementation and the best existing GPU implementation (using the same configuration as our GPU implementation). In Table 3.4 we break down the time spent on each stage of the algorithm for our GPU implementation. For the sequential implementation, we compare against ACOTSP (the source code is available at [53]) which has been established as the standard sequential CPU implementation used when comparing against GPU implementations (see [20, 21]).

We obtained our results by solving the same instances of the TSP as documented by Cecilia et al. [20]. The results for the existing CPU and GPU implementations were obtained by repeating their executions on our hardware to ensure that the comparison is fair and that any speedup is not a product of hardware or software differences. For example, improvements to the CUDA toolkit could inadvertently increase the performance of our implementation which could lead to incorrectly attributing any potential performance improvements to the new algorithm design and not the improvements in the toolkit.

Instance	CPU	Cecilia et al. GPU	Our GPU	Speedup CPU	Speedup GPU
$d198$	48.37	7.60	1.16	41.79x	6.56x
$a280$	123.13	18.61	2.68	45.86x	6.93x
$lin318$	175.57	26.43	3.39	51.79x	7.79x
$pcb442$	482.19	66.34	7.79	61.86x	8.51x
$rat783$	3059.70	332.15	42.70	71.65x	7.78x
$pr1002$	7004.72	646.90	85.11	82.30x	7.60x
$nrw1379$	17711.68	1687.63	323.00	54.83x	5.22x
$pr2392$	97850.65	8026.75	1979.31	49.44x	4.06x

Table 3.3: The Average execution times (ms) of our GPU implementation against the best current GPU implementation from Cecilia et al. and the standard sequential test CPU implementation (ACOTSP [53]).

Instance	Tour Construction	Pheromone Update
$d198$	1.10	0.062
$a280$	2.63	0.059
$lin318$	3.30	0.087
$pcb442$	7.72	0.079
$rat783$	42.56	0.138
$pr1002$	84.96	0.153
$nrw1379$	322.71	0.296
$pr2392$	1977.93	1.380

Table 3.4: The execution times for each stage of our proposed GPU implementation (ms) across various instances of the TSP.

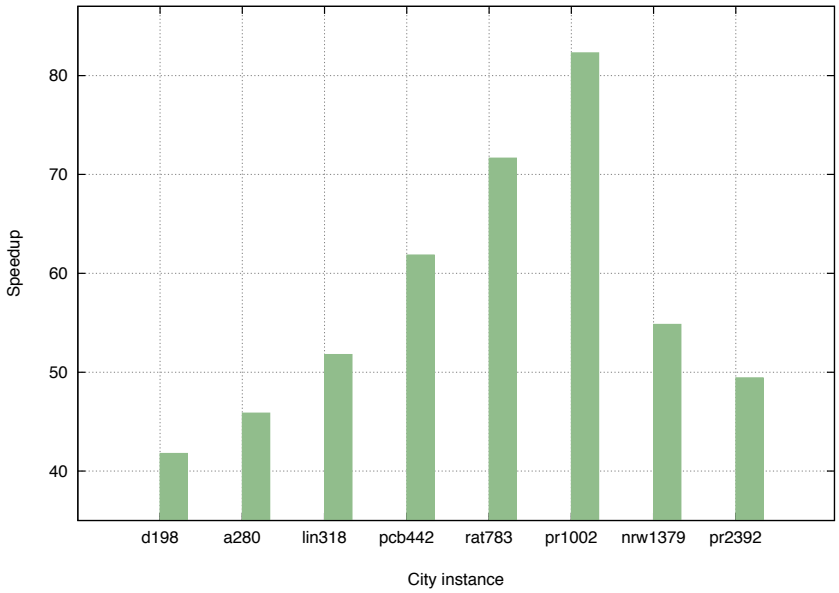


Figure 3.7: Speedup of execution against the standard CPU implementation.

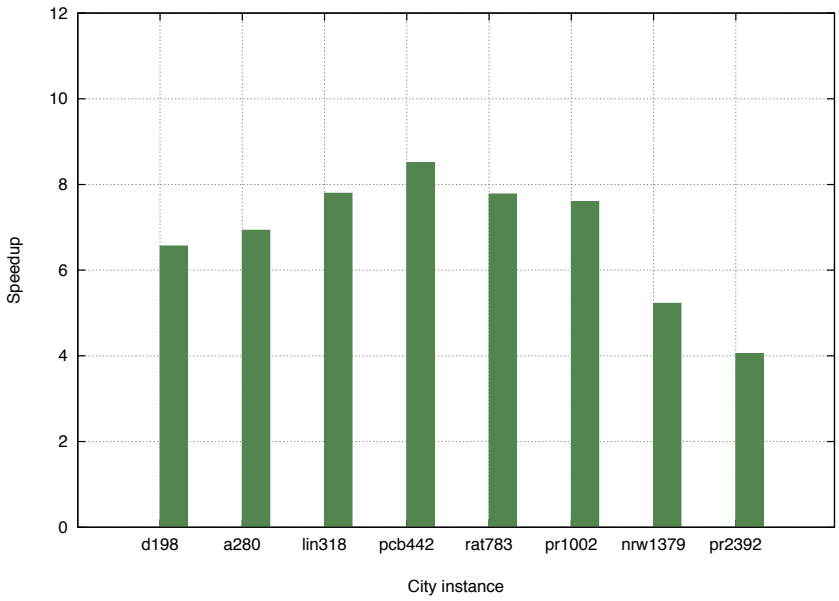


Figure 3.8: Speedup of execution against the best existing GPU implementation.

The results show a speedup of up to 82x faster than the sequential CPU implementation and up to 8.5x faster than the current best parallel implementation. The tour construction stage is responsible for the majority of the total execution time and varies between 4-8.5x faster than the existing GPU implementation. As previously mentioned, the tour construction stage uses a new efficient implementation of roulette wheel selection and we believe our implementation will be able to bring similar speedups to other algorithms limited by the execution time of proportionate selection. The pheromone update implementation is between 1-9x faster than the existing GPU implementation. As the performance of atomic operations is further optimised with subsequent hardware and software releases from NVIDIA, the execution time of the pheromone update stage will decrease further without additional manual software optimisations.

3.6.4 Shared memory restrictions

The size of the shared memory available for each thread block can be reduced (which in turn increases the L1 cache size available) by altering the *preferred cache configuration*. By reducing the shared memory available we observe that for larger tour sizes, the reduction in shared memory size no longer has any affect. From this we can infer that the shared memory in larger instances is exhausted which forces threads to use additional slower global memory and reduces the efficiency of the tour construction phase.

Our parallel roulette wheel selection algorithm improves upon the implementation presented by Cecilia et al. [20] by significantly reducing the amount of shared memory required. As we can observe in Fig. 3.8 as we exhaust the shared memory the effectivness of our solution is reduced and so emphasis was placed on conserving as much shared memory as possible. Our DS-Roulette algorithm only stores the most recently visited city in shared memory (as opposted to the whole tour) and our algorithm doesn't generate as many random numbers which also are stored in shared memory. However, the tabu list is still stored in shared memory which increases in size as the tour size increases. In Chapter 4 we'll explore how to dynamically reduce the size of the tabu list by compressing the tabu list after each city is selected.

Finally the warp-reduce method uses double the shared memory required over a simpler branching implementation and we believe that this contributed to the decreased speedup observed for larger TSP instances (see Fig 3.9). Due to these memory limitations, we did not test any TSP instances larger than 2392 cities.

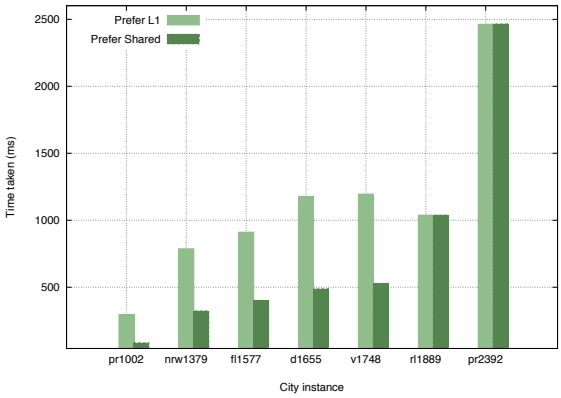


Figure 3.9: Observed execution speeds as a product of varying the L1 cache size.

3.7 Conclusion

In this chapter we have presented a new data-parallel GPU implementation of the AS algorithm that executes both the tour construction and pheromone update stages on the GPU. Our results show a speedup of up to 8.5x faster than the best existing GPU implementation and up to 82x faster than the sequential counterpart. By extending the work of Cecilia et al. [20] and Delévacq et al. [21] we were able to present a new data-parallel approach that focused on utilising each thread block on a warp level to achieve additional speedups over the current best GPU implementation.

For larger data sets we observed that shared memory usage can often be exhausted causing the performance of the algorithm to degrade. Our implementation is able to match the quality of solutions generated sequentially on the CPU. However, we can still improve things further. In the subsequent chapters we detail improvements to reduce the shared memory requirements of DS-Roulette and utilise additional common CPU improvements to the AS algorithm (such as the use of a candidate set to reduce the available search space).

3.7.1 Contributions

Our primary contribution of an efficient parallel implementation of roulette wheel selection contributed significantly to the reported speedups and we envisage the parallel algorithm might be more widely applicable within other heuristic problem-solving areas.

CHAPTER 4

Candidate Set Parallelisation Strategies

This chapter is based on the following publication:
Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU [2].

4.1 Introduction

In this chapter we build upon our data-parallel ACO implementation on the GPU (as described in Chapter 3). We observed that existing parallel implementations of ACO on the GPU [1,20,21,54] fail to maintain their speedup against their sequential counterparts that utilise common CPU optimisation strategies. As the number of cities to visit increases, so does the computational effort and thus time required for the AS algorithm to construct tours as the tour construction phase is performed $N-2$ times to build up a tour. The search effort can be reduced through use of a *candidate set* (or *candidate list*). A candidate set limits the number of potentially valid cities to a smaller subset (often pre-calculated and based on a greedy heuristic such as the closest N cities) when constructing a tour for each move. This chapter is motivated by the shortcomings of existing solutions and we present three candidate set parallelisation strategies for execution on the GPU.

- The first parallelisation strategy adopts a naive ant-to-thread mapping to examine whether the use of a candidate set can increase the performance. This naive approach (in the absence of candidate sets) has previously been shown to perform poorly by Cecilia et al. [20] and Delévacq et al. [21]. However, as neither contribution utilised a candidate set, we first implement this naive approach to test if this simple mapping can be improved when using a candidate set to restrict the search space.

- The second approach extends our previous data-parallel approach (as pioneered by Cecilia et al. [20] and Delévacq et al. [21]), mapping each ant to a thread block on the GPU. Through the use of warp level primitives we manipulate the block execution to first restrict the search to the candidate set and then expand to all available cities dynamically and without excessive thread serialisation.

- Our third parallelisation strategy also uses a data-parallel approach but also compresses the list of potential cities outside of the candidate set in an attempt to further decrease execution time. In Chapter 3 the performance of our parallel implementation decreased as shared memory was exhausted. We aim to address this issue by the use of tabu list compression as the tour is constructed.

We find that our data-parallel GPU candidate set mappings reduce the computation required and significantly decrease the execution time against the sequential counterpart when using candidate sets. By adopting a data parallel approach we are able to achieve speedups of up to 18x faster (see Table 4.2) than the CPU implementation whilst preserving tour quality and show that candidate sets can be used efficiently in parallel on the GPU. As the use of candidate sets is not unique to ACO, we predict that our parallel mappings may also be appropriate for other approaches such as *Genetic Algorithms*.

4.2 Candidate Sets

Randall and Montgomery [55] note that for larger instances of the TSP, the computational time required for the tour construction phase of the algorithm can increase significantly. In Chapter 2 we presented a data-parallel implementation of the AS algorithm for execution on the GPU using NVIDIA CUDA. Our solution was able to outperform the best existing GPU implementations (using CUDA) for all tested instances. However, constructing larger tours still required significantly more time (resulting in a lower speedup against the CPU) due to the increased search space and computational overhead. As the number of cities increases, the number of available moves from any given city during tour construction also increases. The increasing search space necessitates a reduction in the number of valid potential moves for each ant at any given city during tour construction to maintain a viable execution time. Additionally due to the shared memory requirements of DS-Roulette larger TSP instances can exhibit unwanted adverse behaviour, such as register spilling, resulting in a further loss of performance again necessitating additional

modifications to the AS algorithm when using large instances. A common solution to this problem is to limit the number of available cities for each iteration of tour construction using a subset of cities which we refer to as a candidate set. In the following subsections we describe how to use a candidate set when solving the TSP and present the results of enabling the use of a candidate set sequentially against our parallel GPU implementation that does not use a candidate set.

4.2.1 Using candidate sets with the TSP

In the case of the TSP, a *greedy* candidate set contains a set of the nearest neighbouring cities for each city. The number of neighbouring cities is user configurable. This exploits an observation that optimal solutions can often be found by only visiting close neighbouring cities for each city [19] and so can significantly reduce the computational overhead for each iteration during the tour construction phase. For the TSP the candidate set can be generated prior to execution using the distances between cities; this does not change during execution so there is no need to re-evaluate these values. In the tour construction phase (for any given city) the ant will first consider all valid and closely neighbouring cities pertaining to the candidate set. If one or more of the cities in the candidate set has not yet been visited, the ant will apply proportional selection on the closely neighbouring cities to determine which city to visit next from the candidate set. As a result, no cities outside of the candidate set are considered even if they constitute a valid move. If no valid cities remain in the candidate set, the ant then applies an arbitrary selection technique to pick from the remaining unvisited cities (such as moving to the first valid city encountered or simply applying the random proportional rule to all remaining cities as described in Chapter 3). Through the use of a candidate size (with a fixed given size), we can potentially reduce the number of valid cities to consider to a far smaller subset until each city has been visited in the candidate set for each city. Dorigo and Stützle [19] utilise the greedy selection technique to select the closest cities. The sequential ACOTSP implementation used as the standard test implementation in Chapter 3 also uses a simple greedy selection algorithm selecting the closest cities with a variable candidate set size. Randall and Montgomery [55] propose several new dynamic candidate set strategies which can update and change the list dynamically during execution and Reinelt [56] gives an overview of many candidate set selections; however, in this chapter we only use static candidate sets as a static set is the simplest addition we can make to our algorithm to first evaluate if a candidate set can be used efficiently in parallel..

4.2.2 The effect of utilising a candidate set

As previously mentioned the use of a candidate set can significantly reduce the overall execution time by reducing the available search space even when using a sequential implementation. To illustrate this we compare observed speedup of our GPU implementation and the data-parallel GPU implementation presented by Cecilia et al. [20] against the CPU implementation when using a candidate set. In Fig. 4.1 we can observe that the data-parallel implementation of AS presented by Cecilia et al. [20] fails to maintain the speedup previously reported when the standard CPU ACOTSP implementation [53] uses a candidate set. We can observe that all test instances are slower than the sequential counterpart and larger instances (such as pr1002 and pr2392) execute at only 0.2-0.4x the speed of the sequential counterpart. In Fig. 4.2 we can observe that our GPU implementation (outlined in Chapter 3) was able to achieve a speedup against the CPU implementation when using a candidate set. For smaller test instances the speedup ranges from around 5x to 4x and for larger test instances this drops to from around 3x to just 1.25x. In Table 4.1 we give the average execution times for the same range of TSP instances used in Chapter 3 for the CPU implementations (with and without the use of a candidate set), the data-parallel GPU implementation presented by Cecilia et al. [20] and our improved data-parallel implementation presented in Chapter 3.

Although we were able to maintain a speedup against the CPU for all test instances, the viability of using a parallel GPU implementation drops as the tour size increases. Existing solutions were also unable to compete with the CPU implementation when using a candidate set thus necessitating the inclusion of a parallel candidate set on the GPU.

Input	CPU	CPU + CS	Cecilia et al.	Our GPU
d198	48.37	6.39	7.60	1.16
a280	123.13	13.44	18.61	2.68
lin318	175.57	18.60	26.43	3.39
pcb442	482.19	42.37	66.34	7.79
rat783	3059.70	168.90	332.15	42.70
pr1002	7004.72	278.85	646.90	85.11
pr2392	97850.66	2468.40	8026.75	1979.31

Table 4.1: The average execution times (ms) of the CPU implementation (with and without candidate set), the best current GPU implementation from Cecilia et al. and our GPU implementation.

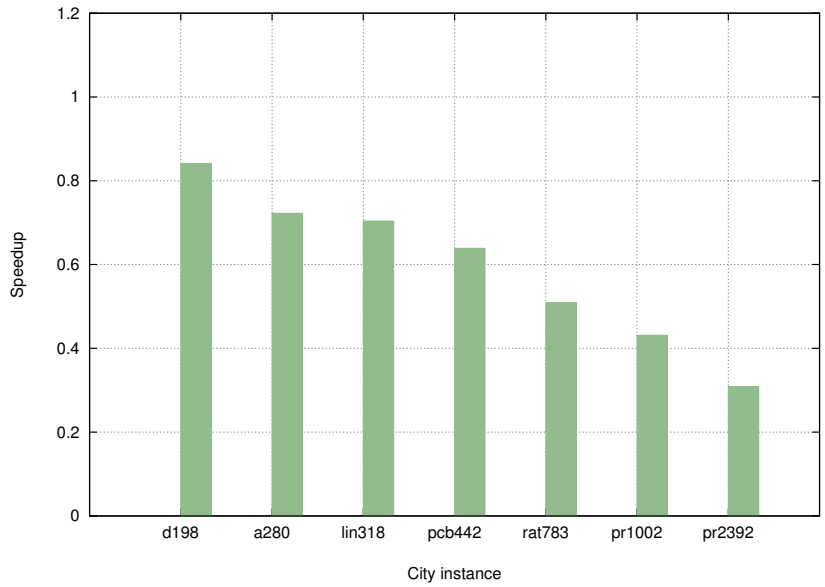


Figure 4.1: Performance of the GPU implementation of AS presented by Cecilia et al. [20] without the use of a candidate set against the sequential CPU implementation [53] using a candidate set.

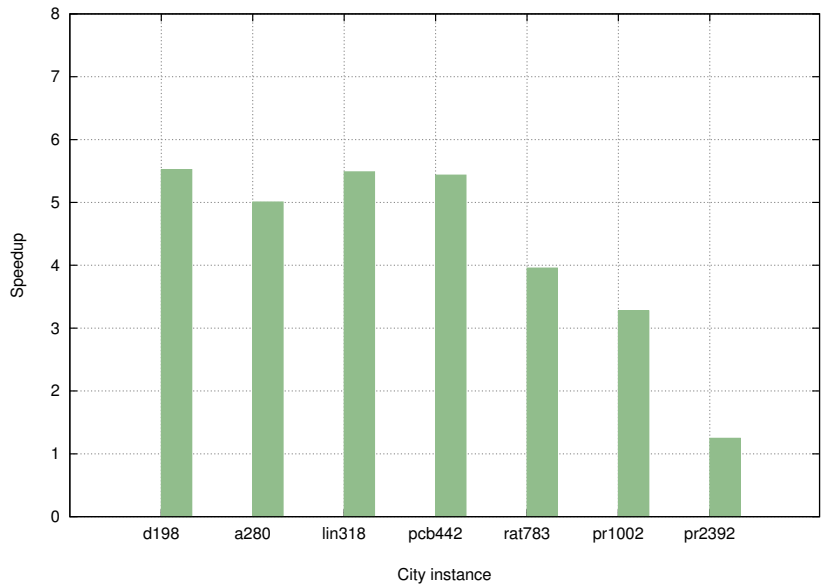


Figure 4.2: Performance of our GPU implementation of AS without the use of a candidate set against the sequential CPU implementation [53] using a candidate set.

4.3 Related work

At the time of writing the best two implementations followed a data-parallel approach and were presented by [20,21]. At around the same time our initial research into ACO on the GPU was published (documented in Chapter 3), Uchida et al. [54] presented a GPU implementation of AS and also use a data-parallel approach mapping each ant to a thread block. Four different tour construction kernels are detailed and a novel city compression method is presented. This method compresses the list of remaining cities to dynamically reduce the number of cities to check in future iterations of tour construction. In existing implementations, all cities are considered when executing the random proportional rule for each iteration of tour construction and visited cities will be assigned a probability of being selected of 0. The selection compression algorithm presented by Uchida et al. [54] maintains an additional unvisited array. This array is then used as the input for the random proportional rule removing the need to check if cities have already been visited and removing them from the selection process. Uchida et al. [54] note that as the number of unvisited cities decreases, the duration of selection process decreases. However, as the global memory reads can no longer be easily coalesced when using the compressed selection process, the performance of the uncompressed variant is greater than that of the compressed variant when the unvisited cities array is large. Additional tour construction methods are proposed, however the performance of all four variants is slower than our data-parallel contribution using the simpler DS-Roulette algorithm using the same hardware and setup as documented in Chapter 3. The speedup reported for their hybrid approach is around 43x faster than the sequential implementation (see [53]). Uchida et al. conclude that further work should be put into nearest neighbour techniques (candidate sets) to further reduce the execution times (as their sequential implementation does not use candidate sets and consequently the performance diminishes when comparing against a CPU implementation utilising candidate sets).

We can observe that the fastest speedups are obtained when using a data parallel approach; however, none of the current implementations [1,20,21,54] use candidate sets and as a result fail to maintain speedups for large instances of the TSP. In conclusion, although there has been considerable effort put into improving candidate set algorithms (e.g. [55,57,58,59]), there has been little research into developing parallel candidate set implementations, thus reducing the viability of existing GPU implementations.

4.4 Implementation

In this section we present three parallel AS algorithms utilising candidate sets for execution on the GPU. The first uses a simple ant-to-thread mapping. Although this approach has previously been outperformed by data-parallel implementations ([1, 20, 21, 54]) we re-examine if this approach is suitable for use with candidate sets. The second and third implementations use a data parallel approach with the third using unvisited city compression to dynamically reduce the size of the unvisited city array to further reduce the computation required during each stage of solution construction. The following implementations will only focus on the tour construction phase of the algorithm. In Chapter 3 we show how to implement the pheromone update efficiently on the GPU [1] and re-use this existing approach for our candidate set implementations.

4.4.1 Basic setup

In Section 3.5 we describe the setup required to initialise the AS algorithm. The majority of these data structures remain unchanged; however, we now include an additional structure for the candidate set. City data is first loaded into memory, stored in an $n \times n$ matrix and transferred to the global memory of the GPU. Ant memory is again allocated to store each ant's current tour and tour length. The ant memory is also copied to the global memory on the GPU. A *pheromone matrix* is initialised on the GPU to store pheromone levels and a secondary structure called *choice_info* is used to store the product of the denominator of Equation 3.1. After initialisation the pheromone matrix is artificially seeded with a tour generated using a greedy search as recommended in [19].

4.4.2 Candidate set setup

The candidate set is represented in memory as a single array of length $n \times c$ (where n is the size of the TSP instance and c is the number of neighbours for each city). To generate the candidate set, we use the city data to save the closest c cities for each city. This simple greedy approach is recommended by Dorigo and Stützle [19] and allows the candidate set to be quickly generated prior to execution. The candidate set is transferred to the global memory of the GPU where it remains unaltered for the duration of the execution. For smaller instances of the TSP this data could also be placed in faster memory types such as texture or constant memory; however, we will only be using the global memory.

4.4.3 Tour construction using a candidate set

In Fig. 4.3 we give the pseudo-code for iteratively generating a tour using a candidate set based upon the implementation by Dorigo and Stützle [19]. First, an ant is placed on a random initial city; this city is then marked as visited in the *tabu* list (located in the shared memory). Then for $n-2$ iterations (again, where n is the size of the TSP instance) we select the next city to visit thus iteratively building a complete and valid tour. The candidate set is first queried and a probability of visiting each closely neighbouring city is calculated. If a city has previously been visited, the probability of visiting that city in future is 0. If the total probability of visiting any of the candidate set cities is greater than 0 (indicating valid cities still remain in the candidate set), we perform roulette wheel (proportional) selection on the set and pick the next city to visit. Otherwise we pick the best city out of all the remaining cities (where we define the best as having the largest current pheromone value that has not yet been visited). Each city is then marked as visited in the tabu list before starting the next iteration of the inner tour construction loop. As the candidate set significantly reduces the search space, selecting the next city to visit from the set is faster than selecting the best city out of the remaining cities.

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

Figure 4.3: Overview of an ant's tour construction when using a candidate set.

4.4.4 Task parallelism

Although it has previously been shown that using a data parallel approach yields the best results [1], [20], [21], [54], it has not yet been established that this holds when using a candidate set. Therefore our first parallelisation strategy considers this simple mapping of one ant per thread (*task parallelism*) to determine if this primitive solution can produce adequate results. Although this would diminish our previous finding, a simpler task parallel candidate set implementation would be advantageous due to the reduced complexity of the solution and thus increase the viability of the algorithm.

Task parallel tour construction

Each thread (ant) in the colony executes the tour construction method shown in Fig. 4.3 and the number of threads is equal to the number ants in the colony. Multiple thread blocks will be used as the number of ants exceeds the number of threads available in a block. There is little sophistication in this simple mapping; however we include it for completeness. Cecilia et al. [20] note that implementing ACO using task parallelism is not suited to the GPU. From our experiments we can observe that these observations still persist when using a candidate set and as a result yield inadequate results which were significantly worse than those obtained by the CPU implementation. We can therefore conclude that the observations made by Cecilia et al. [20] still hold when using candidate sets and the use of a candidate set does not improve the performance enough to warrant using this approach over a data-parallel implementation. As a result our subsequently proposed candidate set parallelisation strategies will all use a data-parallel approach building upon our previous contributions and observations so as to decrease the total execution time of the algorithm by fully exploiting the GPU architecture.

4.4.5 Data parallelism

Our second approach uses a data-parallel mapping (one ant per thread block). Our initial experiments when using the basic task parallel approach (one ant per thread) quickly showed that the previous limitations of the approach remained thus necessitating a different approach. Based on our previous observations made when implementing a parallel roulette wheel selection algorithm [1] we found that using warp level primitives to avoid branching led to the largest speedups. We can also observe that the control flow of DS-Roulette is similar to that of tour construction using a candidate set.

DS-Roulette

In DS-Roulette each warp independently calculates the probabilities of visiting a set of cities (where a set is equal to the number of threads in a warp). These probabilities are then saved to shared memory and one warp performs roulette wheel selection to select the best set of cities. Roulette wheel selection is then performed again on the subset of cities so as to select which city to visit next [1]. This process is fast as we substitute performing reduction across the whole block with two spins of roulette wheel selection and thus avoid waiting for other warps to finish executing and reduce costly synchronisation across the thread block. However, this process is fundamentally different when using a candidate set as we no longer need to perform roulette wheel selection across all potentially available cities. As a result DS-Roulette in its present state is unsuitable for use with a candidate set without significant modifications to initially reduce the number of potential cities to just query the candidate set. Without these modifications all cities would initially be considered when the candidate set should first be queried. Only after the candidate set is exhausted for each city should cities outside of the candidate set be considered.

Reversing DS-Roulette to incorporate the use of a candidate set

In its present state DS-Roulette is unsuitable when utilising a candidate set. DS-Roulette first considers all potential cities to perform roulette wheel on one warp of potential cities. Informally, this process first considers all available cities before restricting to one individual set of cities. When using a candidate set we first perform roulette wheel selection across the candidate set (one set of cities) to potentially select the next city and only scale up to all available cities if no neighbouring cities are available in the candidate set. Informally, this process first considers one set of cities before potentially expanding this to all remaining cities which is the exact opposite control flow of DS-Roulette in its current state. Therefore if we simply reverse the execution path of DS-Roulette (see Fig. 3.4) we can adapt the algorithm to utilise a candidate set during tour construction (see Fig. 4.4) taking advantage of our previous contribution. The process can be further simplified as we can skip the greedy selection process entirely if a valid city is found in the candidate set. This process is significant for larger instances of the TSP as it reduces the execution time as the number of expensive global memory accesses can be decreased when considering the candidate set compared to all the available cities. Our new data parallel tour selection algorithm consists of three main stages which we will now detail.

1. Query the candidate set

The first stage uses one warp to calculate the probability of visiting each neighbouring city in the candidate set. Typically the candidate set size is small and Dorigo and Stützle [19] recommended only using the closest 20 neighbouring cities; however this can change depending on the input size and implementation details. We found experimentally that using a candidate set with less than 32 cities (1 complete warp on CUDA Compute 2.0 and above) was actually detrimental to the performance of the algorithm. This was due to warp serialisation which in turn incurred a large execution speed reduction. Scaling the candidate set up from 20 cities to 32 cities allows all threads within the warp to follow the same execution path and thus avoid costly thread divergence. For larger candidate set sizes, we envision our parallel candidate set implementation based on DS-Roulette could easily be modified to support multiple warps in the first stage of the algorithm. However, to further avoid warp serialisation as a result of divergence, the number of candidate set cities should be a strict multiple of the thread warp size (e.g. 32, 64, 96 and so on).

The first optimisation we apply when checking the candidate set is to perform a warp ballot to quickly check if any valid cities remain in the candidate set before attempting to perform proportional selection. During the warp ballot, each thread in the warp checks the respective candidate set city against the tabu list and submits this value to the CUDA operation __ballot(). The result of the ballot is a 32-bit integer delivered to each thread where bit n corresponds to the input for thread n. If the integer is greater than zero then unvisited cities remain in the candidate set and we proceed to perform roulette wheel selection on the candidate set. Roulette wheel selection is still applied in the first step of the algorithm as we don't want to simply select the closest available city and thus create a purely greedy selection process. Using the same warp-reduce method we previously used in [1] we are able to quickly normalise the probability values across the candidate set warp, generate a random number and select the next city to visit without communication between threads in the warp. Our previously introduced technique for a fast parallel roulette wheel selection in a single warp could easily be applied to other algorithms relying on proportional selection for a section of the algorithm such as genetic algorithms. The city selected will then be used as the next city for the ant to visit whilst building up a complete tour and as a result the remaining two stages for this iteration are skipped entirely. If the result of the warp ballot is equal to 0 we can infer that no valid cities remain in the candidate set and so can skip to stage 2 (bypassing the parallel warp-level proportional selection process) to select the next best available city.

2. Perform a greedy search

The second stage is performed only after we have already checked the candidate set for potential cities and failed to find any available cities. If there were valid available cities in the candidate set, stage 2 and 3 are skipped for the current iteration of tour construction. In stage 2 we dynamically increase the search space to include all available cities for the instance of the TSP. We then progressively narrow down the number of remaining available cities to select the best remaining available city (where best is defined as having the largest pheromone value). We again limit the number of threads per block to 128 and perform tiling across the block to match the number of cities (as previously demonstrated in Chapter 3). Each warp then performs a modified version of warp-reduce [1] to find the city with the highest pheromone value using warp-max. Warp-max quickly finds the largest value within the warp and records the index of the thread so that one nominated thread can save this value. By using a warp level operator we can again avoid additional synchronisation in the warp and unnecessary increases in execution duration. This is a purely greedy operation and so does not require the additional normalisation and random selection stages from roulette wheel selection. As each warp tiles, it saves the current best city and pheromone value to shared memory. As we are solely interested in the best possible available city we do not need to keep the best available city per thread warp and so can override the best discovered value as the warp tiles without consequence. Using this approach we can quickly find four candidates (1 best candidate for each of the warps as there are 128 threads with 32 threads per warp) for the city with the maximum pheromone value for the final stage of the algorithm using limited shared memory and without block synchronisation. There are only four potential cities as each warp only keeps the largest candidate in memory when tiling over all available cities.

3. Select the best available city

The final stage of the algorithm simply uses one single thread to check which of the four previously selected cities has the largest pheromone value, select this city and save the value to global memory and mark as visited in the tabu list. This stage is purely sequential and we have not optimised this for execution in parallel as the computational overhead is so small and occupies a small proportion of the total execution time. However, in future work this step could potentially be merged with step 2 as part of the tiling method when querying all available cities. The three stages of the algorithm are illustrated in Fig. 4.4.

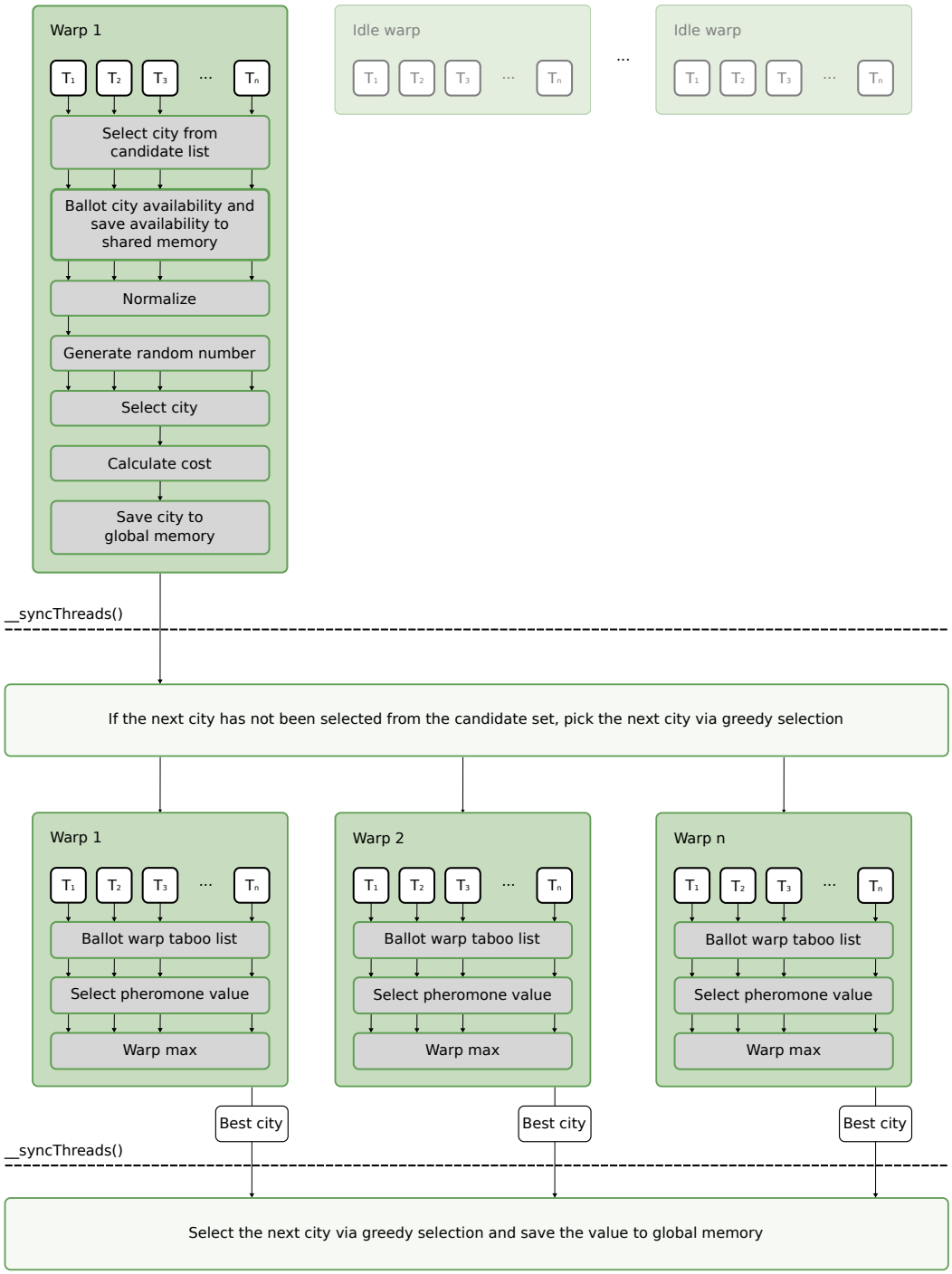


Figure 4.4: An overview of our data parallel tour construction algorithm utilising a candidate set to reduce the total execution time of the algorithm.

4.4.6 Data parallelism with tabu list compression

Section 4.3 details the recent work of Uchida et al. [54] presenting a novel tabu list compression technique when construction tours in parallel on the GPU for the TSP. This novel tabu list compression technique is able to reduce the tour construction execution time. The execution times presented are comparable to our results in Chapter 3. However Uchida et al. do not consider the use of a candidate set and as a result struggle to maintain performance against the sequential implementation when using a candidate set.

Our third candidate set optimisation considers using this novel tabu list compression technique to further reduce the execution time of tour construction. In its current state this process is only useful when considering cities outside of the candidate set. Little benefit can be gained from reducing the search space of the candidate set. As the size of the candidate set is set to match the number of threads in a warp, further limiting the work to a subset of the warp would immediately and clearly result in warp divergence and thus impede the performance of the fast candidate set selection stage. Therefore from the onset we will only consider the use of tabu list compression to stages 2 and 3 of our modified tour construction algorithm (illustrated in Fig. 4.3). In the case where the next city cannot be selected from the candidate set, the search space is subsequently widened and so the use of tabu list compression could potentially limit the number of cities to query and reduce the number of memory calls to the costly global memory.

In our existing GPU solution we represent a tabu list as an array of integers with size n. When a city i is visited, the value tabu[i] is set to 0 which in turn reduces the probability of selecting that city in future iterations of tour construction to 0. However, when using tabu list compression this process is modified to dynamically reduce the size of the tabu list. When city i is chosen, the algorithm replaces city tabu[i] with city tabu[$n-1$] and decrements the list size n by 1. Cities that have previously been visited will not be considered in future iterations thus reducing the search space. By adding tabu list compression to our data parallel tour construction kernel we aim to further reduce the execution time. However, as a complete tabu list is still required for checking against the candidate set we must use two tabu lists and subsequently double the number of global memory accesses. The second list maintains the positions of each city within the first candidate list allowing for quick lookups with a time complexity of $O(1)$. If we were to solely rely upon a single tabu list with compression, the time complexity increases to $O(n)$. As we cannot check if a city in the candidate set has been visited by visiting the specified index in the tabu list, each thread would have to iterate through the entire list.

4.5 Results

In this section we present the results of executing various instances of the TSP on our two data-parallel GPU candidate set implementations. We compare the results of our parallel implementations to the sequential counterpart and our previous GPU implementation (see Chapter 3). As we mentioned in Section 4.4 our task parallel implementation was unable to match the performance of the CPU implementation and offers no practical benefits. As a result we will not include the results of the task parallel trials. From our experiments we would advise against implementing ACO with a candidate set in parallel on the GPU when using a task parallel approach due to the aforementioned limitations (see [20]).

We use the standard ACO parameters as previously used (see Section 3.6) but increase the candidate set size from 20 to 32 to match the warp size (see Section 4.4). The solution quality obtained by our data parallel implementations was able to match and often beat the quality obtained by the sequential implementation. As previously noted the quality of tours obtained by AS are not optimal and can be further improved with local search.

4.5.1 Experimental Setup

For fairness and consistency we used the same experimental setup as described in Section 3.6.1. We used an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield). To again match the setup of Cecilia et al. [20], timing results are averaged over 100 iterations of the algorithm with 10 independent runs.

4.5.2 Benchmarks

In Table 4.2 we present the execution times for a single iteration (where an iteration is defined as a single run of tour construction and pheromone deposit for each ant in the colony) of the tour construction algorithm using a candidate set for various instances of the TSP (these instances were also used in Chapter 3 and by Cecilia et al. [20]). Columns 5 and 6 show the speedup of the two data parallel implementations over the CPU implementation. Both the GPU and CPU implementations are using a candidate set. For a comparison of CPU using a candidate set against GPU implementations without, see Fig. 4.1 and Fig. 4.2. The CPU results are based on the standard sequential implementation ACOTSP (source available at [53]) and the two GPU columns correspond to the two proposed data parallel candidate set implementations in Section 4.4.

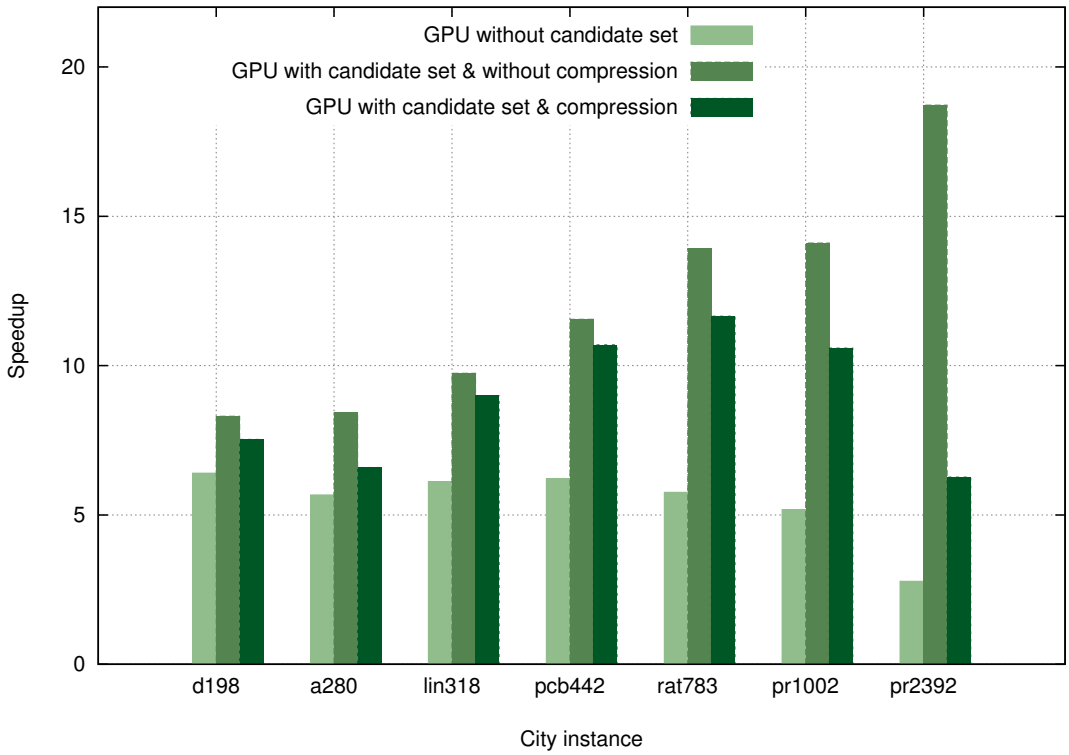


Figure 4.5: A comparison of the speedup of execution of multiple GPU instances (with and without use of a candidate set) against the standard CPU implementation using when using a candidate set.

Instance	CPU	GPU without compression	GPU with compression	Speedup without compression	Speedup with compression
$d198$	6.39	0.77	0.85	8.31x	7.53x
$a280$	13.44	1.59	2.04	8.42x	6.59x
$lin318$	18.60	1.90	2.07	9.74x	8.99x
$pcb442$	42.37	3.67	3.96	11.55x	10.69x
$rat783$	168.90	12.13	14.49	13.92x	11.66x
$pr1002$	278.85	19.76	26.34	14.10x	10.58x
$pr2392$	2468.40	131.85	393.98	18.72x	6.27x

Table 4.2: Average execution times (ms) when using AS and a candidate set.

Our results show that the GPU implementation without compression achieves the best speedups across all instances of the TSP with speedups of up to 18x against the sequential counterpart. Both data parallel approaches consistently beat the results obtained for the sequential implementation for all test instances. The speedup obtained by the GPU implementation without compression increased as the tour sizes increased. This is in contrast to our previous GPU implementation [1], in which the speedup reduced once the city size passed the instance pr1002 due to shared memory constraints, and failed to maintain speedups against the sequential implementation when using a candidate set. By using a candidate set we can avoid these constraints and maintain a speedup for increased tour sizes. The results attained for the GPU implementation with tabu list compression show the implementation was not able to beat the simpler method proposed without compression. As mentioned in Section 4.4, to implement tabu list compression a second tabu list must be used to keep the index of each city in the first list. This second list must be used to avoid costly $O(n)$ lookups to the tabu list over cheaper $O(1)$ lookups. This second list resulted in requiring additional steps during the computationally expensive section of our method and additional calls to global memory. The process of updating the second list for each iteration (for both the greedy search stage and proportionate selection on the candidate set stage) outweighed the benefits of not checking the tabu values for previously visited cities. The increased shared memory requirements for larger instances reduced the performance and effectiveness of the solution.

In Fig. 4.5 we compare the speedup of our previous GPU implementation [1] without a candidate set against our data parallel GPU solutions. We omit comparisons with other data parallel GPU implementations (see Section. 4.3) as we have previously shown our GPU implementation using DS-Roulette to be the fastest to date. We can observe that for large instances of the TSP in Fig. 4.5 the speedup obtained from our GPU implementation without a candidate set reduces whilst the opposite can be seen for our new data parallel implementation without tabu compression.

4.6 Conclusion

In this chapter we have presented three candidate set parallelisation strategies. We have shown that candidate sets can be used efficiently in parallel on the GPU and execute significantly faster than the CPU counterpart. Our results show that a data parallel approach must be used over a task parallel approach to maximise performance. As was

anticipated, a task parallel approach performed poorly and was not able to beat the CPU implementation when using a candidate set. Tabu list compression was shown to be ineffective when implemented as part of the tour construction method and was beaten by the simpler method without compression due to requiring two lists to avoid costly memory accesses. Future work should aim to implement alternative candidate set strategies including dynamically changing the candidate list contents and size.

4.6.1 Contributions

In this chapter our primary contribution has been to highlight that existing GPU implementations fail to compete against the CPU implementation when using common optimisation strategies such as the use of a candidate set. We have shown that a simple extension to DS-Roulette can effectively integrate the use of a candidate set and reduce the overall execution time producing a significant speedup against the CPU counterpart.

CHAPTER 5

Ant Colony Optimization based Image Edge Detection

This chapter is based on the following publication:
Accelerating Ant Colony Optimization based Edge Detection on the GPU using CUDA [3].

5.1 Introduction

In this chapter we extend our data-parallel ACO implementations on the GPU and present the first implementation of a parallel ACO-based image processing edge detection algorithm on the GPU using CUDA. The motivation of this chapter was to improve the runtime of ACO-based edge detection leading to an increased viability of the novel edge detection algorithm, rather than introduce fundamental changes to the design of the algorithm (however, the nuances of CUDA usually mean that algorithm amendments occur so as to secure an efficient implementation). Our implementation is able to match the quality of edge maps produced by existing sequential implementations. By building upon our previous data-parallel approach we are able to successfully execute more ants in parallel per iteration without loss of performance or degradation of the edge map produced. We are able to show that a previously proposed data-parallel ant mapping is still applicable for image processing despite conventional GPU image processing algorithms using the pixel-to-thread mapping (see Section 5.3). The solution presented is able to stay comfortably below 15ms per iteration enabling 60 *frames per second* (FPS).

The primary contribution of this chapter is a novel mapping of individual ants to CUDA thread warps so as to pack multiple ants into a single thread block whilst maintaining the previously proposed data-parallel approach. This approach yields the best results and speedups of up to 150x against an optimised sequential counterpart.

5.2 Edge detection using ACO

ACO has been successfully applied to many different problems including: the TSP [42], the Quadratic Assignment Problem [60] and, in recent years, to a range of image processing problems including edge detection [61]. Dorigo and Stützle remark [19] that for many applications ACO solutions often rival the best in class. Edge detection is the process of producing a map of edges from a given input image. The resulting edge maps are an essential component of many computer vision algorithms as they drastically reduce the input data size whilst preserving vital information concerning the edge boundaries [26]. Baterina and Oppus [62] note that traditional approaches to edge detection can be computationally expensive as an exhaustive search is performed via per-pixel convolution to determine the position of edge boundaries from neighbouring pixels.

Nezamabadi-pour et al. [61] moved away from per-pixel convolution and proposed the first mapping of ACO for edge detection by implementing the AS algorithm. The AS algorithm was previously described in Section 3.2 and outlined in Fig. 3.1. AS consists of two stages: tour construction and pheromone deposit, and is repeated until a termination condition is obtained (such as a set number of iterations or a solution attaining a minimum quality is met). Tour construction in the context of edge detection refers to the decision of which move to take next. An image can be held in memory using a 2D array representation of a graph. This 'grid-based' graph serves as the landscape for each ant to explore. To initialise the algorithm, the input image is read, converted to a graph and each ant is randomly placed on a node. Ants then move around the image following the variations in image intensity. The ants deposit pheromone so as to communicate which edges to follow with other ants in future iterations of the algorithm. Each ant has a limited memory of nodes it is not allowed to visit again so as to ensure that ants do not get stuck following the same trail. Without this memory, an ant could loop around the same set of nodes until termination without exploring other regions of the image. A move is considered valid if the node is not currently in the ant's memory. There is no limitation to how many ants can occupy a single node at any one point (which eliminated the need for communication between ants during solution construction). This process results in ants grouping around the edges of the image which, in turn, is used to produce an edge map (to which a thinning algorithm may be applied). The process produces high quality edge maps but, as Lu and Chen [63] note, overall it can be slow due to inherent redundancy in the search and expensive due to the number of ants.

5.2.1 Solution construction

Once initialised each ant independently moves to one of the eight neighbouring nodes (horizontally, vertically or diagonally). Nezamabadi-pour et al. [61] consider one complete iteration of the algorithm to consist of each ant performing one step (a move from one node to another in the virtual environment) and updating the pheromone matrix for the iteration. In ACO ants decide how to construct solutions using the random proportional rule where each available move is assigned a probability and where the selection of a move is proportionate to the probability [19] (see Section 3.2.1). Nezamabadi-pour et al. [61] adapted the random proportional rule for edge detection by considering two cases. In the first case, they consider that the ant has visited all of the neighbouring nodes and no valid move is currently available: in this case we simply randomly move the ant to another position on the graph thus constituting a move. In the second case (where there are valid moves available), each ant surveys the eight neighbouring nodes (see Fig. 5.1) so as to determine where next to move. The probability of visiting a neighbouring node (i, j) from (r, s) so as to perform a valid move is (for each ant) defined as:

$$p^k_{(r,s),(i,j)} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_u \sum_v [\tau_{uv}]^\alpha [\eta_{uv}]^\beta} \tag{5.1}$$

where: τ_{ij} is the amount of pheromone currently deposited on the pixel (i, j); η_{ij} is the visibility of the node (i, j); and α and β are user-defined parameters to control the influence of τ_{ij} and η_{ij}. The visibility of the pixel (i, j) is defined as:

$$\eta_{ij} = \frac{1}{I_{Max}} \times Max \begin{bmatrix} |I(i-1, j-1) - I(i+1, j+1)|, \\ |I(i-1, j+1) - I(i+1, j-1)|, \\ |I(i, j-1) - I(i, j+1)|, \\ |I(i-1, j) - I(i+1, j)| \end{bmatrix} \tag{5.2}$$

where I is the intensity of a pixel. By applying the simple pixel mask (see Fig. 5.2) to each pixel we can determine the variation in intensities between pixels. Nezamabadi-pour et al. [61] note that edge pixels should have the highest visibility. This in turn directly increases the chance of the pixel being selected by the random proportional rule. As the intensity of the image remains constant throughout execution, the intensity values can be pre-calculated prior to the AS execution to decrease the overall running time.

i-1,j-1	i-1,j	i-1,j+1
i,j-1	i,j	i,j+1
i+1,j-1	i+1,j	i+1,j+1

Figure 5.1: Surrounding neighbour pixels and valid moves from position (i, j) (providing none of the surrounding pixels have recently been visited)

5.2.2 Pheromone update

Once each ant has completed its move (or has been randomly relocated if there were no moves available), the pheromone matrix must be updated. To avoid stagnation of the colony, the pheromone level of every node is first evaporated according to the user-defined *evaporation rate* ρ. So, each pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}. \tag{5.3}$$

Over time this allows nodes that are seldom visited to be forgotten and potentially excluded from the final edge map which is generated from the pheromone matrix. After evaporation the pheromone matrix must be updated with the last move of each ant so as to influence the subsequent iterations of the algorithm. Each ant deposits an amount of pheromone on the last node visited so that the pheromone level τ_{ij} becomes:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \tag{5.4}$$

5.2.3 Termination conditions

The algorithm is executed for a set number of iterations after which an edge map is generated from the pheromone matrix [61]. At this stage it is important to note that the number of iterations is influenced by the number of ants. If we allow more ants to explore the graph simultaneously then we can reduce the total number of iterations required.

5.3 Related work

In this section we will review existing contributions to edge detection using ACO. We will give a brief overview of the most significant contributions to date and of the most significant relevance to our research. At the time of writing we were unable to find any other existing research into parallel edge detection algorithms using ACO with NVIDIA CUDA or using other GPU frameworks (such as OpenCL or older solutions). For completeness we will also briefly cover alternative GPU-accelerated edge detection implementations. We will not include existing parallel ACO TSP research as these have previously been covered and we direct the reader to Chapters 3 and 4.

5.3.1 Edge detection using Ant Colony Optimisation

As was previously mentioned, Nezamabadi-pour et al. [61] were the first to propose edge detection via ACO. Their main contribution was the novel mapping of standard ACO components (random proportional rule and pheromone update) to edge detection using the simple AS algorithm and letting ants explore an artificial landscape based on an input image. The subsequent edge maps produced were of high quality and the input parameters for the algorithm did not require modification for different images. Implementation details and execution times for their algorithm were not included; however, Lu and Chen [63] have subsequently remarked that this initial offering was slow.

Lu and Chen [63] provide an alternative method and focus their efforts on repairing broken edges and reducing the work done by the algorithm. When compared against the implementation by Nezamabadi-pour et al. [61], their results show that they were able to produce higher quality edge maps in around half the time. However, the time required to produce a single edge map was around 1 minute and did not execute in parallel.

Tian et al. [64] detail an improved ACO edge detection algorithm based on the works of Nezamabadi-pour et al. [61]. Their approach differs by allowing ants to make multiple moves on each iteration and to update the pheromone levels after each of these moves and again after all ants have moved. They detail that the execution time of their implementation was also around 1 minute. They conclude that a parallel ACO algorithm could be effectively utilised to decrease the total execution time.

Many additional papers have subsequently been presented however, these papers have mainly focussed on improving the viability of the algorithm by increasing the quality of the edge maps produced over reducing the execution time of the algorithm.

5.3.2 Other edge detection methods on the GPU

The Canny edge detection algorithm [26] produces high quality edge maps and results in more complete edges than alternative algorithms such as Prewitt or Sobel due to the inclusion of the hysteresis step (in which thresholding is applied to include and exclude certain edges). Luo and Duraiswami [27] were the first to present a GPU implementation of Canny using CUDA. Their implementation moved the entire execution of the algorithm to the GPU which yielded speedups over the optimised sequential counterpart. Luo and Duraiswami [27] detail that their GPU implementation uses a pixel-to-thread mapping. Ogawa et al. [28] build upon the work of Luo and Duraiswami [27] and also present a GPU implementation of the Canny algorithm using a simple pixel-to-thread mapping. Simpler edge detection algorithms such as using the Prewitt and Sobel operators have also been implemented in parallel by NVIDIA [65]; the process of applying one of the operators via convolution in parallel is simple and uses pixel-to-thread mapping.

5.4 Implementation

In this section we present a parallel implementation of a ACO edge detection algorithm for execution on the GPU using NVIDIA CUDA. We execute each stage of the algorithm on the GPU to avoid unnecessary memory transfers. Building upon Chapters 3 and 4, we present a new parallel approach mapping multiple ants to each thread block on a warp level. Cecilia et al. [20] have previously shown that mapping individual ants to each CUDA thread is not effective and following a data-parallel mapping of one thread per thread block yields improved execution times. This speedup is due to reducing warp serialisation as a result of eliminating thread divergence caused by each thread following its own path. When a warp is serialised the speedup is dramatically reduced and NVIDIA recommends, as best practice, that this should generally be avoided [30].

5.4.1 Adapting data-parallelism for edge detection

Edge detection differs from the TSP as each ant can potentially move to any city in the graph of size k; however, with ACO based edge detection each ant can only ever move to any of the 8 neighbouring pixels (see Fig. 5.1). The number of valid moves will also decrease as an ant is not permitted to revisit a pixel for a set number of iterations (this is to avoid the ant becoming stuck and not fully exploring the image). To accommodate the

difference in the number of potentially valid moves, we map each ant to a warp of threads and execute multiple ants per thread block. This allows us to ensure all threads within the warp still follow the same execution path (avoiding warp serialisation) and also to execute more ants per thread block thus reducing the total number of blocks required. Whilst we could have simply utilised the previous data-parallel mapping, this would have left most of the threads in each block idle during execution and resulted in increased execution time due to using more thread blocks. As was previously shown by Cecilia et al. [20], mapping a single thread to an individual results in slower execution times and our initial experiments also found this to be true for our edge detection implementation. As a result the following section will only document our warp-level ant mapping which resulted in the best speedups and lowest execution times for all of the test images.

5.4.2 Algorithm setup

As Nezamabadi-pour et al. [61] note, an input image can be loaded into a 2D array. The color image is then converted to greyscale using the algorithm shown in Fig. 5.2. Novak and Shafer [22] note that around 90% of edges in an image can be found just using the greyscale values and this approach produces high quality edge maps.

The image array is later used by the random proportional rule for determining the visibility of a pixel which in turn alters the probability of an ant deciding to move to the pixel. However, as this input image remains static throughout each iteration of the algorithm we can significantly reduce the computational load of the algorithm by pre-processing the pixel visibilities. The visibility of a pixel is calculated using the variation of intensity around a given point. With existing implementations each ant must calculate the visibility of all pixels in the neighbourhood of a pixel for each iteration. This is a costly operation and unnecessary as the image data does not change. After loading the image data into an array, we calculate all pixel visibilities and save the results to a second array in global memory on the GPU. This array is then used by each ant when calculating the probability of visiting a neighbouring pixel thus replacing eight slow global memory lookups with a single lookup for each pixel.

```
procedure ColorToGreyscale (red, green, blue)
    return (red >>2) + (green >>1) + (blue >>2);
end
```

Figure 5.2: Calculating the greyscale value for a color pixel

A pheromone matrix is allocated in global memory on the GPU and artificially seeded with the value 0.0001 as described by Nezamabadi-pour et al. [61]. A second pheromone matrix is also allocated in global memory. As each ant is executed in parallel, it can potentially deposit pheromone to the pheromone matrix before all ants have made their next move. To accommodate this, after evaporation the new values are written to the second matrix. Each ant then deposits an amount of pheromone on the second matrix after constructing their solution using an atomic add operation. In the next iteration of the algorithm the two pheromone matrices are swapped thus allowing ants to deposit without impacting the current iteration. Finally we allocate an array containing the ants. For each ant we maintain the current position on the image, the current iteration and a small array for the ant memory. As we are operating on the warp level we set the length of the memory array to 32 previous locations matching the size of the warp. By maintaining the current iteration, we are able to treat the memory array as a circular array using basic modulo arithmetic. For example on iteration 48 we would index position 48 % 32. This allows us to maintain a fast FIFO queue of previous locations on the image without the need for additional data structures. Before we enter the solution construction phase, the ants are randomly placed around the image. Lu and Chen [63] suggest an alternative to randomly placing the ants is to place the ants on the end points of edges extracted using alternative algorithms. However we will use the random placement for our implementation described by Nezamabadi-pour et al. [61].

5.4.3 Solution construction

In Section 5.2 we gave an overview of solution construction phase of the AS algorithm outlined by Nezamabadi-pour et al. [61]. In this section we describe our parallel mapping of the algorithm to the GPU using CUDA. Our primary contribution of this chapter is the novel ant-to-warp mapping for the tour construction phase. As was previously mentioned, mapping a single ant to a thread is ineffective and leads to warp serialisation resulting in a loss of performance. Mapping a single ant to each thread block is a wasteful use of the GPU as most threads will be idle for each iteration. As a result we cannot simply use DS-Roulette. We found experimentally that packing four ants into a thread block (using a total of 128 threads) and operating on a warp level yielded the best results. In Fig. 5.3 we illustrate our ant warp mapping for solution construction that is key to the speedup attained. In Fig. 5.4 we give an overview of the entire parallel solution construction phase performed by each ant within a thread block before detailing each step individually.

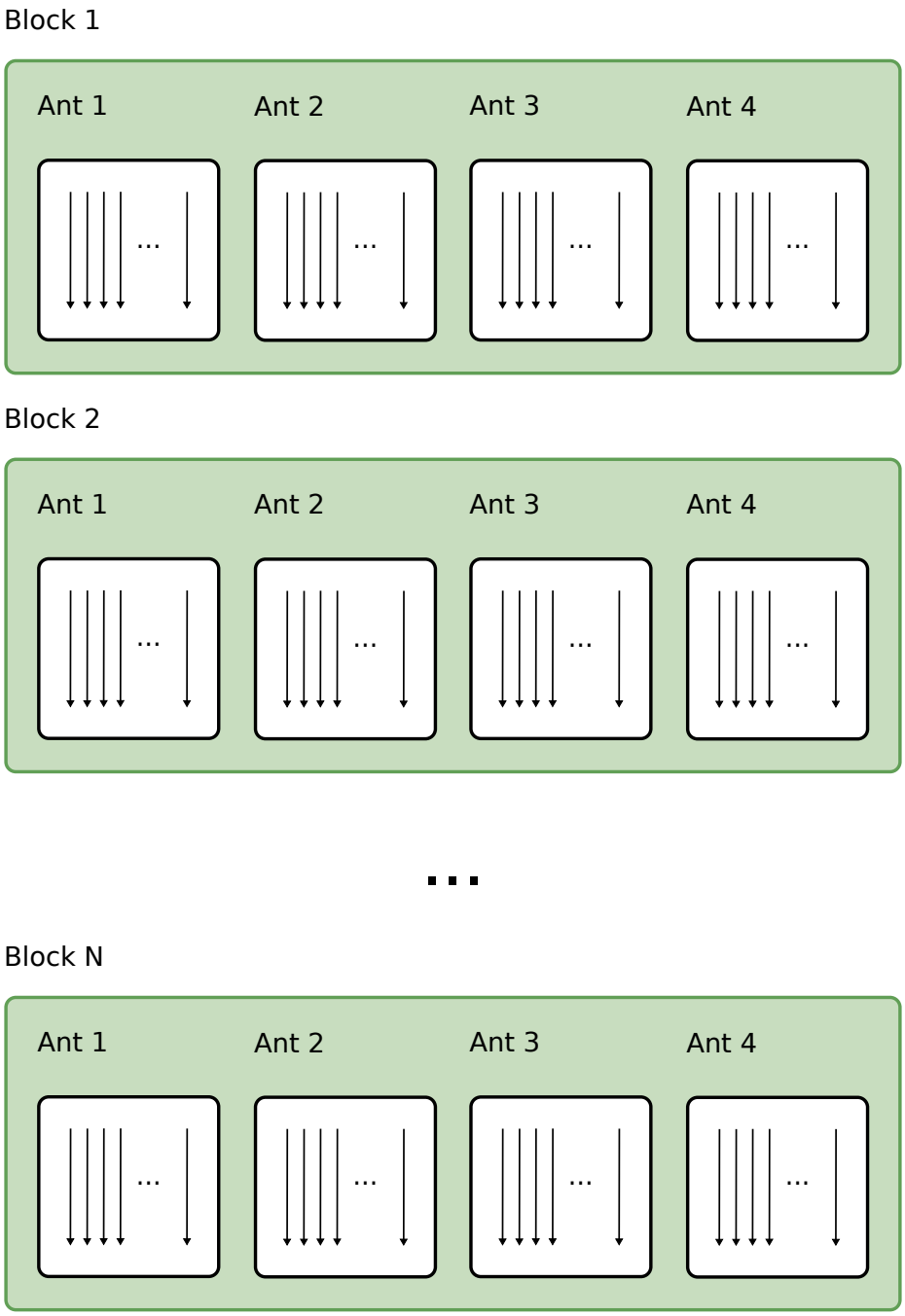


Figure 5.3: An overview of our novel mapping technique to pack multiple ants within a thread block. Each ant is mapped to a thread warp and four warps are executed per thread block in parallel.

```
procedure SolutionConstruction
   Cache the surrounding pixel visibility data
   Calculate the probability of visiting local pixels
   Check if the local pixels have already been visited
   Perform reduction on the pixel probabilities
   Perform roulette wheel selection on the probabilities
   Update the ants current position
end
```

Figure 5.4: An overview of solution construction

Cache the surrounding pixel visibility data

First the pre-calculated visibility values (as previously described) of the 8 neighbouring pixels are cached into shared memory. As we are operating on a warp level, the remaining 24 threads load in the same data to avoid warp divergence. As the values are cached in the L1 cache, this operation is faster than branching the warp. After the visibility data is cached to shared memory, each thread loads the previously visited position from the ant's memory into a local register. As the size of the memory is limited to 32 previous positions each thread will load one value from the array. For example, thread 9 will load position 9 from the array and so on. This will later allow the warp to quickly check if a position has recently been visited without touching costly global memory.

Calculate the probability of visiting local pixels

Each of the first 8 threads in the warp then calculate the probability of visiting one of the neighbouring pixels using the random proportional rule described in Section 5.2. The probabilities calculated are saved to an array in shared memory, again to avoid the cost of using global memory. Although we introduce branching in this stage, having the remaining 24 threads perform additional work is more costly than serialisation.

Check if the local pixels have already been visited

Each thread in the warp then checks whether the previously visited pixel cached in its register is a valid move. If a thread finds that the move has recently been made it replaces the probability of visiting that pixel with 0 (this matches our previous implementation in Chapter 3 replacing available cities with potentially available neighbouring pixels).

Perform reduction on the pixel probabilities

A warp-level reduction is then performed on the probability array returning the total of all the probability values. At this point the first thread checks if the total value returned is greater than 0 and if not the thread picks a new random location for the ant to move to (also saving this to the ants memory). If the total is greater than 0 the first thread then calculates a random number in the range of 0 to 1, saving this to shared memory.

Perform roulette wheel selection on the probabilities

The first 8 threads then apply roulette wheel selection (proportionate selection) to select the next pixel to move to. Using the previously reduced probability array in shared memory (and total value previously returned) each thread normalises the probability of its respective pixel bringing the probability into the range of 0 to 1 (see Table 3.1). After the values are normalised each of the 8 threads then checks if the previously generated random number lies within the range for that pixel. If the thread determines that its pixel has been selected then the ant's current position is updated. This parallel implementation thus handles both cases for solution construction outlined by Nezamabadi-pour et al. [61].

5.4.4 Pheromone update

The second stage of the AS algorithm is pheromone update which consists of two stages: pheromone evaporation; and pheromone deposit. The pheromone update stage represents a very small proportion of the total execution time and thus is not the main focus of this paper. As we previously noted [1], the pheromone evaporation stage (see equation (3.3)) is trivial to parallelise as all points on the matrix are evaporated by a constant factor ρ. We adopt the same efficient parallel strategy previously demonstrated [1] where a single thread block is launched which maps each thread to a position on the pheromone matrix and decreases the value using ρ. A tiling strategy is used to ensurecoverage of all positions on the matrix. Each ant then deposits an amount of pheromone proportional to the quality of its move onto the pheromone matrix. As previously noted, we utilise two pheromone matrices (alternating between primary and secondary) to ensure that ants deposit to a matrix not currently being read by other ants. Each ant deposits an amount of pheromone to the second pheromone matrix using the operation *atomicAdd()* (for thread safety to avoid overriding values) which takes the previous value within the matrix and adds the new value.

5.5 Results

In this section we will discuss our experimental setup, algorithm parameters chosen, quality of the edge maps obtained via our parallel implementation and finally the execution times observed from our implementation on the GPU against the CPU counterpart.

5.5.1 Experimental setup

For fairness and consistency we used the same experimental setup as described in Chapter 3 and again in Chapter 4. This identical setup includes an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield).

5.5.2 Algorithm parameters

To ensure a fair comparison of the edge maps produced via our implementation, we use the same parameters as defined by Nezamabadi-pour et al. [61] with the exception of modifying the number of ants and iterations. Nezamabadi-pour et al. define the number of ants and iterations to be proportionate to the root of the size of the image. For example, an image of size 512×512 would have a total of 512 ants performing 512 iterations before producing an edge map (a total of 262144 moves). We found experimentally that executing more ants per iteration for fewer iterations yielded edge maps of comparable quality and was a better fit for the parallel architecture as we can execute more ants simultaneously with ease. There is an implicit overhead when scheduling a kernel for execution on the GPU and by increasing the number of ants (and thus increasing the number of thread blocks) we can reduce the number of kernel executions. This approach will also scale automatically to CUDA devices with more CUDA cores which will further decrease the execution time. For a 512×512 image we use 3000 ants for a maximum of 50 iterations (a total of 150000 moves). The remaining algorithm parameters are as follows:

- $\alpha = 2.5$
- $\beta = 2$
- $\rho = 0.04$
- Ant memory length = 32
- Edge threshold = mean image intensity

5.5.3 Solution quality

To ensure our implementation provided edge maps comparable to the original algorithm by Nezamabadi-pour et al. [61], we tested against the standard test images (*Lena, Peppers etc.*). To evaluate what we consider to be a good edge map we have to consider a number of factors including; the thickness of the edges produced, if edges are repeated (i.e. double edges), if edges are incorrectly detected, if the edges that are detected are incomplete and if the edges we have detected have been incorrectly shifted. Our parallel solution was able to match and often improve the quality of edge maps produced. In Fig. 5.5 we show the edge maps produced by the Sobel operator, the Canny edge detector and our parallel ACO implementation. The edge maps produced for Canny and Sobel were generated via the Image Processing toolbox in MATLAB [66].

(a) Original (b) Sobel

(c) Canny (d) Our solution

Figure 5.5: A comparison of final edge maps produced by the Sobel, Canny and our parallel ACO edge detection algorithms.

5.5.4 Variable edge thickness

We found experimentally that by increasing the number of ants (whilst keeping the iteration count constant) we were able to increase the thickness of the edges produced. Over time ants settle on the major edges in the image due to deposits on the pheromone matrix. As more ants are added to the image this effect is amplified creating a stylised effect. In Fig. 5.6 we show the edge maps produced when using 1500 ants, 3000 ants (standard), 4500 and 6000 ants.

(a) 1500 ants (b) 3000 ants

(c) 4500 ants (d) 6000 ants

Figure 5.6: The effect on edge thickness when alternating the number of ants in a 512x512 standard test image.

5.5.5 Benchmarks

In Section 5.4 we detailed the various mappings for ACO on the GPU. The simplest of the mappings uses a single CUDA thread for each ant, the second mapping uses a whole CUDA thread block for each ant and the third mapping uses a single thread warp per ant (with multiple ants per CUDA block). Our results (shown in Table 5.1) show that the different approaches yield significantly different results. The execution times detailed are for both the solution construction and pheromone update stages of the AS algorithm.

Threads per block	Ant-to-thread	Ant-to-block	Ant-to-warp
64	62.679	21.435	10.693
128	63.025	18.233	6.531
256	62.858	18.433	8.884
512	62.601	18.431	9.308
768	64.845	18.973	9.597

Table 5.1: Average execution times (ms) when varying the number of threads per block and mapping arrangement of ants to blocks, threads and warps.

The first mapping (although the simplest to implement) produced the worst results and varying the number of threads per block did little to change this. As Cecilia et al. [20] note, this simple mapping is not suited to ACO as the solution construction phase results in warp divergence which increases the overall execution time. As expected the second data-parallel mapping produced significantly better results and executed in around a third of the time of the first mapping. The third approach which utilised our ant-to-warp mapping consistently produced the best results. The mapping performed best when using 128 threads (4 ants using 4 warps of 32 threads per CUDA thread block).

In our ant-to-warp implementation each thread in the warp caches a value of a recently visited city to shared memory. This later allows the warp to quickly check if a neighbouring pixel is still valid or has been visited recently without touching the slower global memory and slowing down the execution. As Fermi generation GPUs automatically cache values to the L1 cache, we observed the execution times of manually caching values to the shared memory versus solely relying on the automatic L1 cache (see Fig. 5.7). The results show that manually caching the values is still considerably faster and necessary to obtain the best possible speedups although requiring additional programming effort.

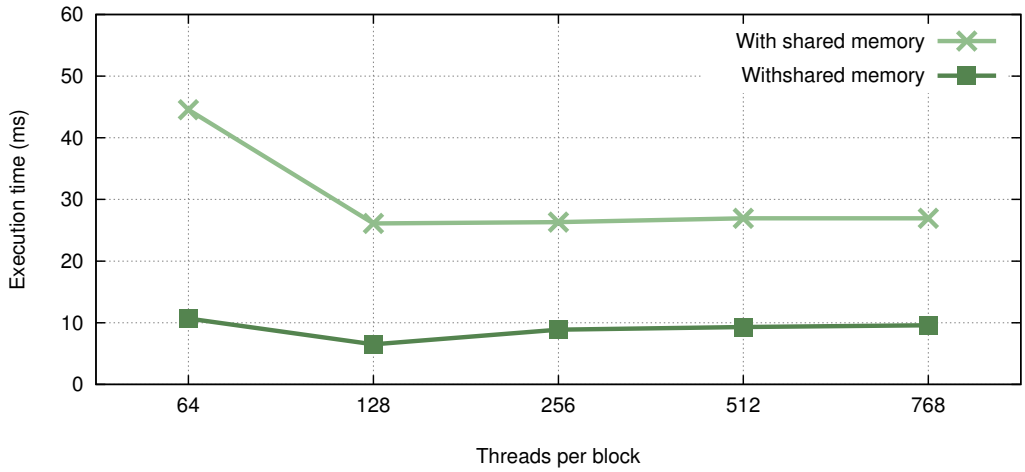


Figure 5.7: Observed execution speeds for the ant-to-warp mapping with/without caching visited positions to shared memory.

An ant placed on an input image has no perceivable concept of difficulty and will explore an image irrespective of the number of edges on the image. As a result we found that unlike with other edge detection algorithms (such as the Canny edge detector) varying the input image had little effect on the overall execution time of the algorithm.

Finally, we compared the results of executing our ant-to-warp mapping implementation against an optimised sequential counterpart. We found that the sequential implementation took just under 1000ms to execute compared against 6.531ms for the best thread configuration for our parallel CUDA version. This represents around a 150x speedup for our GPU implementation against the optimised CPU implementation.

5.6 Conclusion

In this paper we present the first parallel ACO edge detection implementation for execution on the GPU. By extending our previous contributions in Chapters 3 and 4 we are able to adapt our fast data-parallel GPU ACO mapping for edge detection using a novel ant-to-warp mapping. By harnessing the massively parallel nature of the GPU we reduced the number of iterations required to produce the edge map and increased the number of ants per iteration. Our implementation is able to match the quality of edge maps produced by the sequential implementation and executed up to 150x faster.

CHAPTER 6

Color Image Edge Detection

This chapter is based on the following joint publication:
Color Image Edge Detection based on Quantity of Color Information and its Implementation on the GPU [4].
Y. Xiang initially proposed utilising additional color information to enhance the performance of edge detection. Y. Xiang and L. Dawson jointly designed the proposed technique and the parallel implementation was proposed and implemented by L. Dawson.

6.1 Introduction

In Chapter 5 we introduce the first-data parallel implementation of an existing established ACO-based edge detection technique on the GPU using NVIDIA CUDA. In this chapter we present a new method for quantifying color information as to detect edges in color images. Novak and Shafer [67] note that although 90% of edge images can be detected using just gray values in color images. The remaining 10% of edges in color images may contain important information for further processing. As a result several color edge detection algorithms have been proposed [23, 24, 68, 69]. Our novel approach uses the volume of a pixel in the HSI color space, allied with noise reduction, thresholding and edge thinning to produce an edge map that can withstand great levels of noise in images whilst creating high quality edge maps. In order to improve the viability of our color edge detection algorithm, we also present a parallel implementation for direct execution on the GPU using NVIDIA CUDA. Experimental results show that compared to traditional edge detection methods our new method can improve the edge detection accuracy, withstand significantly greater levels of noise on the input algorithm and achieve speedups of up to

10x over related CUDA implementations based on the Canny edge detection algorithm using CUDA. Although our color edge detection method is not as sophisticated as the method presented in Chapter 5, we envisage our new method could be successfully applied to detect edges in noisy images whilst maintaining a high frame rate in domains such as real-time video processing or when using poor quality image sensors.

6.2 Color image edge detection

In Chapter 5 we introduce edge detection, a fundamental step in computer vision, image processing, analysis and pattern recognition systems. Its importance arises from the fact that edges often provide an indication of the physical extent of objects within the image. They are often vital clues toward the analysis and interpretation of image information. By the detection of edges the size of the image data is reduced into a size that is more suitable for image analysis. As later tasks (such as image segmentation, boundary detection, object recognition and classification, image registration, and so on) depend on the success of the edge characterization step, it is very important that the edge detection step should provide sufficient information to characterize the image feature, but with a relatively small image size (to keep the image edge information). Hence, edge detection must be efficient and reliable [70].

To utilize the non-linear relationship between color components, we propose a new color edge detection algorithm based on the quantity of color information (QCI) in the HSI color space. To detect as many edges as possible, we use QCI as well as the existing luminance values used by traditional edge detection techniques. Not all edges can be detected by QCI, but some edges not detected by luminance information can be detected by QCI. Experimental results show that our algorithm generally performances as good as the Canny edge detection algorithm [26], and in some cases, it performances better and can endure more noise than the Canny edge detection algorithm.

6.2.1 Edge detection techniques

An edge in a monochrome image is defined as an intensity discontinuity, while in a color image, the additional variation in color must be considered; for example, the correlation among the color channels (which may constitute a valid edge). There are a number of color edge detection methods which can be divided into two groups: 1. techniques extended from monochrome edge detection; 2. vector space approaches, including vector

gradient operators, directional operators, compound edge detectors and second derivative operators [71, 72]. Numerous kernels have been proposed for finding edges; for example, the Prewitt kernels [73] and Sobel [74] kernels. By using gradient operators the edges are detected by looking for the maximum in the first derivative of the color or intensity of the images. Second derivative operators search for zero crossings in the second derivative of the color or intensity of the image to find edges [75]. First derivative operators are very sensitive to noise, while the second derivative operators lead to better performance in a noisy environment. After selecting a suitable color space, primary edge detection steps include: (1) suppressing noise by image smoothing; and (2) localizing edges by determining which local maxima correspond to edges and which to noise (thresholding). A Gaussian filter is widely used to remove noise. The Gaussian operator is isotropic and therefore smoothes the image in all directions [76]. One problem with derivative-based edge detection is that the output may be thick and require edge thinning [77].

6.3 Related work

In this section we will give an overview of the most significant contributions made to edge detection using CUDA. In Section 5.3.2 we outlined significant contributions to edge detection using CUDA; however, will expand upon these here. At the time of writing we were unable to find any other existing research into parallel color edge detection algorithms using NVIDIA CUDA.

Luo and Duraiswami [78] present a parallel version of the Canny edge detection algorithm using CUDA. Their work extends upon a previous parallel contribution on the GPU presented by Fung [79]; however, it uses CUDA as opposed to the older NVIDIA Cg framework. Their contribution represents the first modern CUDA implementation of the Canny edge detection algorithm and is able to improve upon an optimised sequential counterpart. The implementation presented by Luo and Duraiswami also improves upon the earlier work of Fung by increasing the functionality to include the hysteresis thresholding step on the GPU. Sections of the implementation such as Gaussian blurring were built upon NVIDIA SDK examples and can be easily reused for other potential applications. The results of [78] show a modest speedup compared to highly optimised sequential Intel SSE code executed on the CPU and significant speedups compared to sequential naive Matlab code also executed on the CPU. However it is worth noting that their code was tested using first generation CUDA hardware and not optimised to take

advantage of the newer features of Fermi, Kepler and Maxwell cards. Luo and Duraiswami note that hysteresis thresholding occupies over 75% of the overall runtime. This is due to the function to connect edges between thread blocks being called multiple times and without this stage the algorithm performs around 4 times faster. They note that the runtime can vary depending on the complexity of the image and number of edges.

Building upon the work of Luo and Duraiswami, Ogawa et al. extend the implementation of the Canny edge detection algorithm [80]. They note that as the hysteresis step is called a fixed number of times, some edges which span over blocks may not be fully traversed. Their implementation solves this by introducing a stack onto which weak edges are pushed and when all edges have been traversed, the algorithm will terminate. Their results show a speedup of around 50 times for large images; however it is unclear as to whether they first compare with niave CPU code or SSE optimised code. As with the implementation presented by Luo and Duraiswami [78], it would be reasonable to assume that the runtime of their improved method will also vary depending on the image. The implementations of Luo and Duraiswami and Ogawa et al. only support gray input and to date, there have been no color edge detection algorithms implemented using CUDA.

NPP (NVIDIA Performance Primitives) is a closed-source CUDA library targeted specifically at video and image processing (although the scope is set to increase with time). NPP allows developers to easily port existing sections of Intel Performance Primitives (IPP) C/C++ code to corresponding GPU functions. We will not be using the NPP library, as it is not an open source library and modifications cannot be made.

6.4 Implementation

In this section we will present the steps to our color edge detection algorithm and also how we map our new algorithm to the GPU using CUDA for parallel execution. Our color edge detection algorithm is divided into 4 steps.

Step 1. Noise removal and color space transformation. The following Gaussian filter is used to remove noise.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{6.1}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution.

We choose to use the HSI (Hue-Saturation-Intensity) color space as it represents colors similarly to how the human eye senses colors, and it is one of the most commonly used color spaces in image processing. For a given color P in the HSI color space, H (hue) is given by an angular value ranging from 0 to 360 degrees, S (saturation) is given by a radial distance from the cylinder center line, and I (intensity) is the height along the cylinder axis. The following three formulas convert a color in RGB (R, G, B) to a color (H, S, I) in the HSI color space:

$$I = \frac{R + G + B}{3} \tag{6.2}$$

$$S = 1 - \frac{min(R + G + B)}{I} \tag{6.3}$$

$$H = \begin{cases} \theta & \text{if } G \geq B \\ 2\pi - \theta & \text{otherwise} \end{cases} \tag{6.4}$$

$$where \quad \theta = arccos \left[\frac{\frac{1}{2}[(R - G) + (R - B)]}{\sqrt{(R - G)^2 + (R - B)(G - B)}} \right] \tag{6.5}$$

Step 2. Gradient finding. In this step we detect edges using the magnitude of the volume information and intensity information.

Let $P = Img(i, j)$ be a pixel at position (i, j) in the image Img, and its color information be (H_p, S_p, I_p).

Define its volume $V_p = \pi \times S_P^2 \times I_p \times (H_p/360)$. As well as volume information, color intensity information is also considered in our algorithm. Replace each pixel's color information in Img by the corresponding volume (resp. intensity) information, and denote it as Img_V (resp. Img_I).

Using Prewitt kernels we calculate the magnitude of the volume and intensity information. Let Img_P be the 3×3 area centered at pixel P in Img_V (resp. Img_I), and let $G_x = D_x \times Img_p$ and $G_y = D_y \times Img_p$ be the magnitude in the x direction and the y direction for P in Img_V (resp. in Img_I), where D_x is the x directional Prewitt kernel, and D_y is the y directional Prewitt kernel. The magnitude of the volume information (resp. intensity information) for the pixel P is defined as: $M1_p = \sqrt{G_x^2 + G_y^2}$.

Another two direction operators are applied in our algorithm to obtain more accurate edges: one in the 45° direction; and one in the 135° direction. Similarly, the magnitude of the volume (resp. intensity) information is defined as $M2_p = \sqrt{G_{45^\circ}^2 + G_{135^\circ}^2}$. A pixel is considered as on an edge if either $M1_p$ or $M2_p$ is above a corresponding threshold.

Step 3. Thresholding. The threshold value is set as the average volume magnitude value multiplied by a constant. For the constant, we first set a random value and then adjust until we get a satisfiable edge map. As a result different images will have a different constant value and this will have to be configured per image.

Step 4. Edge thinning. It is neccessary to thin the edges of the edge map produced due to using a derivative-based edge detection algorithm. Prior to thinning, the edges may be up to several pixels thick thus requiring thinning. For our algorithm we use a simple edge thinning algorithm described by Guo and Hall [81] (an extension of [82]).

6.5 Parallel Implementation on the GPU

In this section we describe how the four steps of the algorithm described in Section 6.4 are mapped to the GPU using NVIDIA CUDA.

6.5.1 Algorithm setup

Image data is stored in the CUDA vector type *uchar*4 which allows us to access the individual RGB components of a pixel easily without additional bit shifting techniques. The *uchar*4 vector is the same size as a standard integer so introduces no additional space constraints by switching to this efficient data type. By using the vector type *uchar*4 global memory accesses can be coalesced for maximum bandwidth as described in step 1 and 2.

For simplicity our solution is currently designed for square input images so as to easily fit the block and thread requirements of CUDA. However, our solution could easily be extended to allow for non-square input images by setting padding on the images. This padding would be applied as the image is loaded so that the image appears as a square internally and thus not requiring any further alterations to the parallel kernels.

Our solution is designed to utilise advancements made in CUDA Compute 2.0 such as increased shared memory and full warp thread processing (pior to Compute 2.0 threads were addressed by half warp). As a result of these hardware requirements, our solution is incompatible with older devices without changing the execution flow of warps and incurring execution penalties due to the lack of the larger L1 cache.

6.5.2 Edge detection

Step 1. Gaussian blurring and color space transformation. The first step of our algorithm applies a Gaussian blur to the input image whilst preserving the three color channels. We initially set the standard deviation $\sigma = 1$ for the Gaussian filter. In our implementation this is a user configurable value and this can be altered at run time to change the final edge map produced. As applying a Gaussian filter is a simple operation, we base our implementation on the optimised image convolution CUDA example provided by NVIDIA as part of the CUDA SDK [83]. In the example provided by NVIDIA, the Gaussian filter is expressed as the product of two one-dimensional filters. Podlozhnyuk [84] describes how using separable filters can increase image convolution efficiency as the number of pixels loaded into shared memory can be reduced. When performing convolution on an image, surrounding image data is required for each pixel and splitting the Gaussian filter into the two separable planes minimises the number of surrounding pixels required which can lead to significant performance benefits [84]. Threads at the edges of thread blocks will require additional pixel information from neighbouring blocks in order to perform convolution. Podlozhnyuk [84] describes how this can be handled by loading an apron of surrounding pixels. Without sampling the surrounding pixels around thread blocks, edges cannot be traced across multiple thread blocks without producing discontinuation of the edges discovered. Luo and Duraiswami [78] also use the efficient apron-based approach described by Podlozhnyuk [84] for their parallel Canny edge detection algorithm.

The seperable convolution example only supports blurrinsg of gray images and as a result is only optimised for a single channel. For a color image the three channels can be blurred independently and recombined after the last blur. This basic approach would allow us to use the source provided without modification (requiring three passes, one for each image channel), however, by combining the three iterations into one single pass, we can take advantage of the pixel data existing in shared memory.

At the end of the second convolution pass (we perform two convolution passes as convolution has deconstructed into the two constituent dimensions) we calculate the color space transformation as again, the pixel data already exists in shared memory thus removing unnecessary accesses to slow global memory. After color space transformation, the results are written back to global memory. An optimisation tested was to replace the inverse cosine function (used as part of the color space transformation) with a faster approximation method; however, this approximation function performed worse than the compiler optimised fast-math alternative and was not used.

Step 2. Gradient calculation. For the next step of the algorithm we calculate the gradient through convolution with the Prewitt operator. As we apply the operator in 4 directions (two of which are diagonal) the process is not linearly separable, as with the blur operator, and requires a different approach to achieve efficient convolution. Stam [85] describes how peak convolution performance can be attained by using shared memory efficiently and gives insight into how this method could be modified to support color images across three channels. This solution also maps multiple pixels per thread to achieve higher device occupancy and reduce the number of apron pixels loaded. However, as the example provided uses gray images, the shared and global memory access patterns are carefully aligned for smaller data types. Based on a modified version of the solution proposed by Stam [85] we perform a fast efficient 2D convolution of the kernel that handles larger float datatypes and multiple color channels. In Figures 6.1, 6.2 and 6.3 we show how image data is loaded into shared memory efficiently to maximise coalesced accesses. Shared memory is aligned across 32 memory banks thus satisfying the access pattern requirements and avoiding bank conflicts. In tests we found that our approach was around 2-3 faster than naive global lookups on Compute 2.0. Once the convolution is complete, the values are again saved back to the global memory.

Step 3. Average gradient calculation and thresholding. In order to perform global edge thresholding the average gradient values must first be calculated. Using the results obtained from the previous convolution we implement a parallel reduction algorithm based on the example provided by NVIDIA [86]. As CUDA does not support global synchronisation the algorithm is split into two stages. The first stage calculates the sum within each block; the second stage then calculates the global total across all blocks. Before saving the total average value to global memory, the value is multiplied by a constant to obtain the thresholding value.

With the global threshold values calculated, the next stage is the relatively simple task of applying thresholding to each pixel. We can process each pixel independently removing the need for any apron areas surrounding the blocks. The first thread of each block then caches the previously computed threshold value into shared memory using a single 128-bit load. Each thread then loads the corresponding pixel data into local registers and performs a conflict-free broadcast access to the cached threshold. Each thread then decides if the pixel loaded qualifies as an edge and writes this data back to global memory. At this point there is an option to either display the edges as they are, or continue to process the data and thin the detected edges.

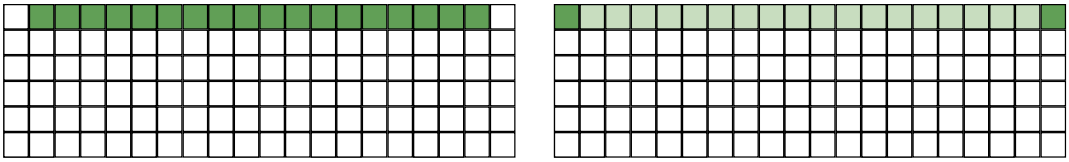


Figure 6.1: Load the top apron pixels

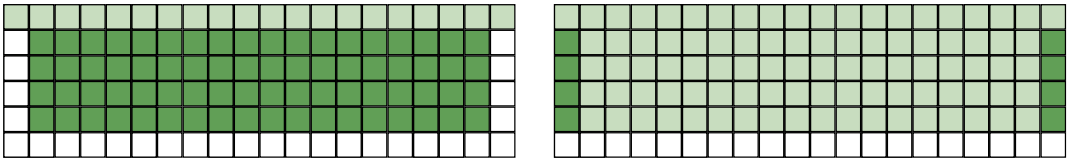


Figure 6.2: Load the centre and apron pixels

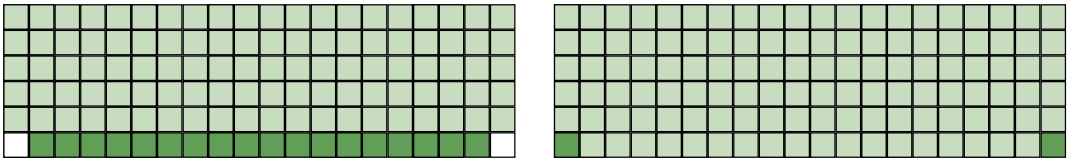


Figure 6.3: Load the bottom apron pixels

Step 4. Edge thinning. Using the edge thinning algorithm mentioned in Section 6.4, we are able to thin the edges in the edge map produced in Step 3. We can avoid the costly task of calculating if each pixel satisfies multiple conditions in order to be thinned by pre-calculating the outcome of any possible situation and storing this data in a lookup table (LUT). As there are only 9 pixels in each window of interest, each of which can be 0 or 1, there are only 512 different outcomes of the edge thinning algorithm per pixel. The outcome of any pixel's calculation can be quickly and efficiently accessed in the LUT by calculating the index addressed according to the matrix shown in Fig. 6.4.

256	32	4
128	16	2
64	8	1

Figure 6.4: LUT index matrix

For example if all pixels are set, the lookup index will be $256+32+4+128+16+2+64+8+1$. This is equal to the index 512 in the lookup table. The value from the LUT is then assigned to the pixel and repeated for all other pixels. This is a commonly used optimisation in image processing and is possible in this circumstance due to the small kernel size. To perform edge thinning in parallel, the results of thresholding are loaded into shared memory and each thread calculates the index for the LUT for each pixel. The final step copies the value obtained from the LUT back into global memory resulting in a thinner edge. This process can be repeated multiple times for thinner edges.

6.6 Results

In this section, we describe the quality of the edge maps produced using our algorithm and timings from the parallel implementation of the GPU, and compare our algorithm with other edge detection algorithms. For each edge detection algorithm the results shown are based on the criteria of selecting the best output by tuning the corresponding parameters. This is required as each algorithm requires fine tuning to produce the optimum edge map for a given input image. To evaluate our proposed technique, we compare our algorithm with the Canny and Sobel edge detection algorithms for a set of color images of the standard image processing dimension 512×512.

6.6.1 Noise tolerance

To demonstrate our algorithms' tolerance to noise we present the results of experiments using the standard test image Lena. In Fig. 6.5 the images in column one are: original, +20% noise, and +40% noise. Columns two to four contain the results of our algorithm, Canny and Sobel, respectively. With no noise we can see that our algorithm detects more useful edges than Sobel and produces an edge map comparable to Canny.

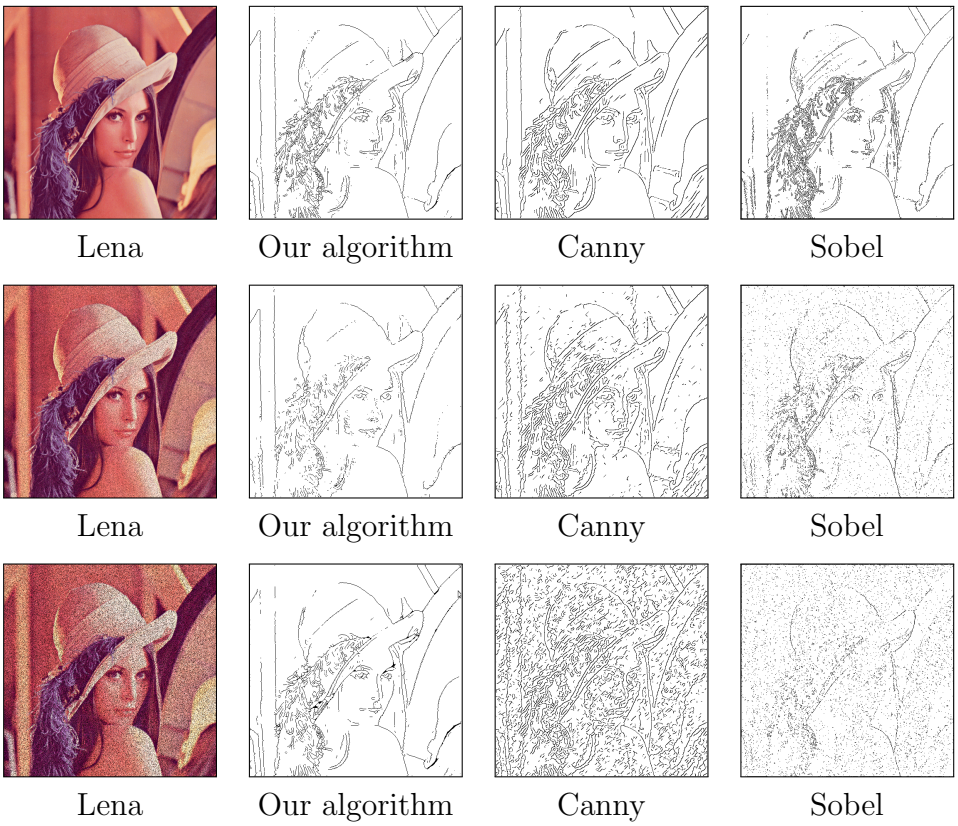
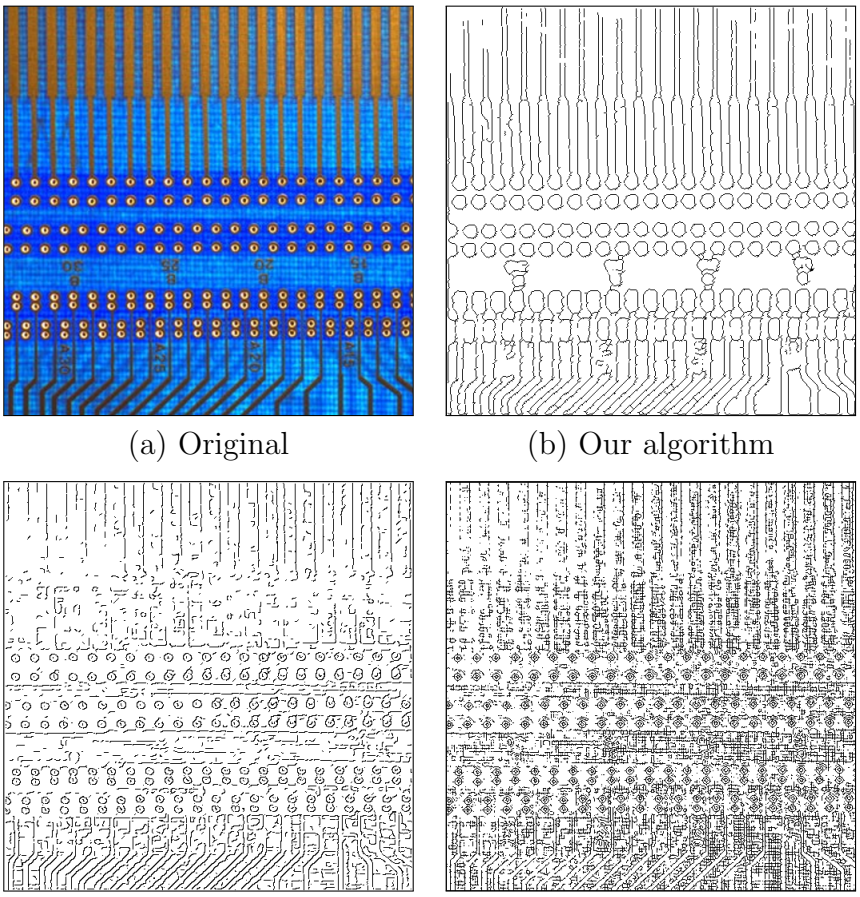


Figure 6.5: The effect of introducing noise to the input image when using the standard image processing test image *Lena*.

When noise is introduced to the image, our algorithm maintains a near constant edge map whilst Canny and Sobel begin to falter. When noise is added, Canny and Sobel required significant input parameter changes whilst our algorithm required minimal input changes. We can observe that the hysteresis stage of the Canny edge detection algorithm begins to form incomplete edges due to an increased threshold to remove noise.

6.6.2 Improved edge detection

By using coefficients for intensity and volume, in some circumstances we are able to produce better edge maps using this volume information. For example when using the test image shown in Fig. 6.6 the edge maps produced are significantly better than the edge maps produced using Canny and Sobel. In this example we adjusted the parameters to allow more volume information than intensity to contribute to the edges. Experimentally we found that our algorithm was often able to match the edge map produced by Canny for most test images. However, the edge maps produced contained thicker edges and sometimes non-connected edges.



(a) Original (b) Our algorithm

(c) Canny (d) Sobel

Figure 6.6: Edge maps produced for a color image.

6.6.3 GPU Implementation Results

In this section we present the results of executing our parallel implementation on the GPU using CUDA. We compare against five standard test images (Lena, Peppers, Mandril, House and Cameraman [87]) and average our results over 1000 iterations of our algorithm.

6.6.4 Experimental setup

For fairness and consistency we used the same experimental setup as described in Chapter 3, Chapter 4 and finally in Chapter 5. This identical setup includes an NVIDIA GTX 580 GPU (Fermi) and an Intel i7 950 CPU (Bloomfield).

6.6.5 Benchmarks

We compare the execution time of our algorithm in parallel against the GPU CUDA Cannny implementation and the OpenCV Canny implementation. As the results from the Canny CUDA implementation were obtained using Compute 1.0 hardware, we tested and re-calculated the results using our updated setup to provide a more accurate and fair comparison. We also executed the OpenCV implementation on our hardware to again provide a fair comparison. Running the hysteresis step four times as proposed by Luo and Duraiswami [78] was rarely sufficient and we also include timings from executing the step 10 times, producing a more accurate Canny output edge map.

Image	OpenCV	GPU Canny (H4)	GPU Canny (H10)	Our algorithm
Lenna	7.95	1.46	2.41	0.46
Mandril	9.05	1.43	2.34	0.44
Peppers	8.12	1.39	2.29	0.47
House	7.78	1.35	2.69	0.44
Cameraman	8.15	1.42	2.53	0.42

Table 6.1: Average execution times (ms) for processing standard image processing test input images using the OpenCV canny implementation, the CUDA GPU implementation [78] with 4 passes of the hysteresis step, the CUDA GPU implementation [78] with 10 passes of the hysteresis step and finally using our new color edge detection method.

6.7 Conclusion

In this chapter we propose a new edge detection method based on QCI in HSI color space. By using the QCI, we are able to find edges that are not detectable when only intensity information is used. Experimental results show that our proposed algorithm is able to withstand greater levels of noise than other leading edge detection algorithms. We present a parallel implementation of our algorithm on the GPU using NVIDIA CUDA. Our GPU implementation can achieve speedups of up to 20 times over the IPP Canny algorithm, and is around 5 times faster than the existing CUDA Canny algorithm.

CHAPTER 7

Conclusion

In thesis we introduce NVIDIA CUDA and present novel parallel mappings of existing and new algorithms on the GPU using NVIDIA CUDA. Our research primarily focussed on improving parallel Ant Colony Optimisation on the GPU and then extended to parallel edge detection using the techniques and mappings outlined in Chapter 3.

In Chapter 3 we presented a new data-parallel GPU implementation of the AS algorithm that executes both the tour construction and pheromone update stages on the GPU. By extending recent contirbutions [20, 21] we adopted a data-parallel approach that focused on utilising each thread block on a warp level to achieve additional speedups over the current best GPU implementations. We presented a new efficient parallel implementation of roulette wheel selection called DS-Roulette that we envisage will be more widely applicable within other heuristic problem-solving areas. By improving the tour construction stage of the algorithm, our novel parallel implementation of roulette wheel selection was able to significantly increase the performance of tour construction and contributed to a speedup of up to 8.5x faster than the best existing GPU implementation and up to 82x faster than the sequential counterpart.

In Chapter 4 we show that to solve large instances of the TSP using AS, the use of a candidate set is essential to maintain a speedup over the sequential counterpart when using a candidate set. We show that all existing contributions struggle to maintain a speedup against a sequential implementation when a canddiate set is used. By improving upon our implementation presented in Chapter 3 we again solve the TSP using three distinct approaches that all utilise a candidate set. After implementing and testing the three candidate set parallelisation strategies we were able to show that candidate sets can be utilised efficiently in parallel when using a data-parallel approach. As was predicted, a

task-parallel approach performed poorly as was not able to beat the CPU implementation when using a candidate set. Our best approach was able to achieve a speedup of up to 18x against the CPU counterpart using a candidate set.

In Chapter 5 we presented the first implementation of a parallel ACO-based edge detection algorithm on the GPU using CUDA. By further extending and reusing our data-parallel approach presented in Chapter 3. we mapped individual ants to thread warps allowing multiple ants to execute per CUDA thread block. Our efficient parallel mapping was able to execute significantly more ants per iteration whilst maintaining a speedup of up to 150x against the sequential counterpart. We hope that reducing the execution time of an ACO-based implementation of edge detection will increase its viability in image processing and computer vision.

In Chapter 6 we outlined a new method of edge detection using additional color information when using the HSI color space. We presented a novel GPU implementation utilising an efficient edge-thinning algorithm that was able to execute up to 20x faster than an efficent CPU implementation of the Canny edge detection algorithm and up to 5x faster than the GPU implementation. Our new method was able to find edges that are not detectable when only using intensity information and crucially was able to withstand significantly greater levels of noise on the input images.

The reusable techniques and mappings presented in Chapter 3 allowed us to improve upon the best existing parallel implementations. These techniques were then extended in Chapter 4 and repurposed in Chapter 5 showing the versatility of our implementation.

7.0.1 Future work

In future work we would like to apply these techniques and mappings to other heuristic problem-solving areas that rely heavily on block synchronisation and would benefit from warp-level parallel programming. As parallel implementations of genetic algorithms have already been implemented on the GPU [88, 89] we would like to see how our technique could be merged with existing approaches.

As the amount of global memory increases on the GPU we would like to see larger instances of ACO processed in parallel. By also extending our work to execute over multiple graphics cards we would like to see huge instances of the TSP such as the Mona Lisa [90] challenge being attempted in parallel. New techniques to split tours over multiple thread blocks would have to be devised along with some method of recombining these tours.

Bibliography

[1] L. Dawson and I. A. Stewart, "Improving Ant Colony Optimization performance on the GPU using CUDA," in *IEEE Congress on Evolutionary Computation, IEEE-CEC'13, Cancun, Mexico, June 20-23, 2013*, pp. 1901–1908, 2013.

[2] L. Dawson and I. A. Stewart, "Candidate Set Parallelization Strategies for Ant Colony Optimization on the GPU," in *Algorithms and Architectures for Parallel Processing*, vol. 8285 of *Lecture Notes in Computer Science*, pp. 216–225, 2013.

[3] L. Dawson and I. A. Stewart, "Accelerating Ant Colony Optimization based Edge Detection on the GPU using CUDA," in *IEEE Congress on Evolutionary Computation, IEEE-CEC'14, Beijing, China, July 6-11, 2014*, pp. 1901–1908, 2014.

[4] J. Zhao, Y. Xiang, L. Dawson, and I. A. Stewart, "Color Image Edge Detection based on Quantity of Color Information and its Implementation on the GPU," in *23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'11)*, pp. 116–123, 2011.

[5] D. B. Kirk and W. H. Wen-mei, *Programming Massively Parallel Processors. A Hands-on Approach*. Newnes, 2012.

[6] NVIDIA, "8800 FAQ."
`http://www.nvidia.com/object/8800_faq.html`
(last accessed 18/07/2014).

[7] A. Rege, "An Introduction to Modern GPU Architecture."
`ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`
(last accessed 07/07/2015).

[8] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[9] NVIDIA, "CUDA for ARM Platforms is Now Available."
`http://docs.nvidia.com/cuda/cuda-c-best-practices-guide`
(last accessed 18/07/2014).

[10] NVIDIA, "CUDA for ARM Platforms is Now Available."
https://developer.nvidia.com/content/cuda-arm-platforms-now-available
(last accessed 18/07/2014).

[11] A. Humber, "Tokyo Tech Builds First Tesla GPU Based Heterogeneous Cluster To Reach Top 500," 2008.
http://www.nvidia.com/object/io_1226945999108.html
(last accessed 15/07/2015).

[12] NVIDA, "ChinasS investment in GPU supercomputing begins to pay off big time!," 2011.
http://blogs.nvidia.com/blog/2011/06/09/chinas-investment-in-gpu-supercomputing-begins-to-pay-off-big-time/
(last accessed 15/07/2015).

[13] NVIDA, "GeForce Grid Press Presentation," 2012.
http://assets.sbnation.com/assets/1119979/GeForce_Grid_Press_Presentation.pdf
(last accessed 15/07/2015).

[14] NVIDA, "Maxwell Architecture," 2014.
https://developer.nvidia.com/maxwell-compute-architecture
(last accessed 15/07/2015).

[15] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, vol. 6, pp. 4464–4466, Oct 2007.

[16] B. Wilbertz *et al.*, "GPGPUs in computational finance: Massive parallel computing for American style options," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 837–848, 2012.

[17] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC bioinformatics*, vol. 9, no. Suppl 2, p. S10, 2008.

[18] Thrust, "Thrust."
https://github.com/thrust/thrust
(last accessed 18/08/2014).

[19] M. Dorigo and T. Stützle, *Ant Colony Optimization.* MIT Press, 2004.

[20] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldon, "Enhancing data parallelism for Ant Colony Optimization on GPUs," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.

[21] A. Delèvacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel Ant Colony Optimization on Graphics Processing Units," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 52–61, 2013.

[22] C. Novak and S. Shafer, "Color edge detection," in *Proc. DARPA Image Understanding Workshop*, vol. 1, pp. 35–37, 1987.

[23] B. Bouda, L. Masmoudi, and D. Aboutajdine, "Cvvefm: Cubical voxels and virtual electric field model for edge detection in color images," *Signal Process.*, vol. 88, no. 4, pp. 905–915, 2008.

[24] C. Lopez-Molina, H. Bustince, J. Fernandez, P. Couto, and B. D. Baets, "A gravitational approach to edge detection based on triangular norms," *Pattern Recognition*, vol. 43, no. 11, pp. 3730 – 3741, 2010.

[25] P. Melin, O. Mendoza, and O. Castillo, "An improved method for edge detection based on interval type-2 fuzzy logic," *Expert Systems with Applications*, vol. 37, no. 12, pp. 8527 – 8535, 2010.

[26] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[27] Y. Luo and R. Duraiswami, "Canny Edge Detection on NVIDIA CUDA," in *Computer Vision and Pattern Recognition Workshops (CVPRW'08)*, pp. 1–8, IEEE, 2008.

[28] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny Edge Detection Using a GPU," in *First Int. Conf. Networking and Computing (ICNC*, pp. 279–280, IEEE, 2010.

[29] R. Farber, *CUDA Application Design and Development.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.

[30] NVIDIA, "CUDA C Programming Guide."
`http://docs.nvidia.com/cuda/cuda-c-programming-guide`
(last accessed 23/08/2014).

[31] NVIDIA, "Tuning CUDA Applications for Kepler."
`http://docs.nvidia.com/cuda/kepler-tuning-guide`
(last accessed 23/08/2014).

[32] NVIDIA, "CUDA Pro Tip: Do The Kepler Shuffle."
`http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle`
(last accessed 10/09/2014).

[33] Dr. Dobs, "Atomic Operations and Low-Wait Algorithms in CUDA."
`http://www.drdobbs.com/parallel/atomic-operations-and-low-wait-algorithm/240160177`
(last accessed 19/10/2014).

[34] K. Group, "OpenCL."
https://www.khronos.org/opencl
(last accessed 23/08/2014).

[35] Mark Harris, "Maxwell: The Most Advanced CUDA GPU Ever Made."
http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made
(last accessed 19/09/2014).

[36] David Patterson, "The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges."
http://www.nvidia.co.uk/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf
(last accessed 19/10/2014).

[37] NVIDIA, "Whitepaper: NVIDIA GeForce GTX 980."
http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF
(last accessed 08/10/2014).

[38] NVIDIA, "CUDA for ARM Platforms is Now Available."
http://devblogs.nvidia.com/parallelforall/cuda-arm-platforms-now-available
(last accessed 18/07/2014).

[39] ARM, "ARM Processor Architecture."
http://www.arm.com/products/processors/instruction-set-architectures/index.php
(last accessed 18/07/2014).

[40] ARM, "NEON."
http://www.arm.com/products/processors/technologies/neon.php
(last accessed 18/07/2014).

[41] ARM, "big.LITTLE Processing."
http://www.arm.com/products/processors/technologies/biglittleprocessing.php
(last accessed 18/07/2014).

[42] M. Dorigo, *Optimization, Learning and Natural Algorithms.* PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy, 1992.

[43] A. Catala, J. Jaen, and J. Modioli, "Strategies for accelerating ant colony optimization algorithms on graphical processing units," in *IEEE Congress on Evolutionary Computation (CEC)*, pp. 492–500, Sept. 2007.

[44] W. Jiening, D. Jiankang, and Z. Chunfeng, "Implementation of ant colony algorithm based on GPU," in *Sixth Int. Conf. on Computer Graphics, Imaging and Visualization (CGIV)*, pp. 50–53, Aug. 2009.

[45] J. Fu, L. Lei, and G. Zhou, "A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection," in *Third Int. Workshop on Advanced Computational Intelligence (IWACI)*, pp. 260–264, Aug. 2010.

[46] W. Zhu and J. Curry, "Parallel ant colony for nonlinear function optimization with graphics hardware acceleration," in *Proc. Int. Conf. on Systems, Man and Cybernetics*, pp. 1803–1808, Oct. 2009.

[47] H. Bai, D. Ouyang, X. Li, L. He, and H. Yu, "MAX-MIN ant system on GPU with CUDA," in *Fourth Int. Conf. on Innovative Computing, Information and Control (ICICIC)*, pp. 801–804, Dec. 2009.

[48] You, Y.S., "Parallel ant system for traveling salesman problem on GPUs." `http://www.gpgpgpu.com/gecco2009` (last accessed 19/10/2014).

[49] W. W. Hwu, *GPU Computing Gems Emerald Edition.* Morgan Kaufmann, 2011.

[50] M. A. Oneil, D. Tamir, and M. Burtscher, "A parallel gpu version of the traveling salesman problem," in *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 348–353, 2011.

[51] K. Rocki and R. Suda, "An efficient GPU implementation of a multi-start TSP solver for large problem instances," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pp. 1441–1442, ACM, 2012.

[52] W. W. Hwu, *GPU Computing Gems Jade Edition.* Morgan Kaufmann, 2011.

[53] M. Dorigo, "Ant Colony Optimization - Public Software." `http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html` (last accessed 29/01/201).

[54] A. Uchida, Y. Ito, and K. Nakano, "An efficient gpu implementation of ant colony optimization for the traveling salesman problem," in *Networking and Computing (ICNC), 2012 Third International Conference on*, pp. 94–102, 2012.

[55] M. Randall and J. Montgomery, "Candidate set strategies for ant colony optimisation," in *Proceedings of the Third International Workshop on Ant Algorithms*, ANTS '02, (London, UK, UK), pp. 243–249, Springer-Verlag, 2002.

[56] G. Reinelt, *The traveling salesman: computational solutions for TSP applications.* Springer-Verlag, 1994.

[57] M. Deng, J. Zhang, Y. Liang, G. Lin, and W. Liu, "A novel simple candidate set method for symmetric tsp and its application in max-min ant system," in *Advances in Swarm Intelligence*, pp. 173–181, Springer, 2012.

[58] H. M. Rais, Z. A. Othman, and A. R. Hamdan, "Reducing iteration using candidate list," in *Information Technology, 2008. ITSim 2008. International Symposium on*, vol. 3, pp. 1–8, IEEE, 2008.

[59] A. Blazinskas and A. Misevicius, "Generating High Quality Candidate Sets by Tour Merging for the Traveling Salesman Problem," in *Information and Software Technologies*, pp. 62–73, Springer, 2012.

[60] V. Maniezzo and A. Colorni, "The Ant System applied to the Quadratic Assignment Problem," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 5, pp. 769–778, 1999.

[61] H. Nezamabadi-pour, S. Saryazdi, and E. Rashedi, "Edge Detection using Ant Algorithms," *Soft Computing*, vol. 10, no. 7, pp. 623–628, 2006.

[62] A. V. Baterina and C. Oppus, "Image Edge Detection Using Ant Colony Optimization," *WSEAS Transactions on Signal Processing*, vol. 6, no. 2, pp. 58–67, 2010.

[63] D.-S. Lu and C.-C. Chen, "Edge detection improvement by ant colony optimization," *Pattern Recognition Letters*, vol. 29, no. 4, pp. 416–425, 2008.

[64] J. Tian, W. Yu, and S. Xie, "An ant colony optimization algorithm for image edge detection," in *IEEE Congress on Evolutionary Computation (IEEE-CEC'08)*, pp. 751–756, IEEE, 2008.

[65] NVIDIA, "Image Convolution with CUDA." `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf` (last accessed 10/01/2014).

[66] MathWorks, "Image Processing Toolbox." `http://www.mathworks.co.uk/products/image/` (last accessed 10/01/2014).

[67] C. Novak and S. A. Shafer, "Color edge detection," in *Proc. DARPA Image Understanding Workshop*, pp. 35–37, 1987.

[68] F. Arandiga, A. Cohen, R. Donat, and B. Matei, "Edge detection insensitive to changes of illumination in the image," *Image Vision Comput.*, vol. 28, no. 4, pp. 553–562, 2010.

[69] Y. Liu, T. Ikenaga, and S. Goto, "An mrf model-based approach to the detection of rectangular shape objects in color images," *Signal Process.*, vol. 87, no. 11, pp. 2649–2658, 2007.

[70] D. Ziou and S. Tabbone, "Edge detection techniques - an overview," *International Journal of Pattern Recognition and Image Analysis*, vol. 8, pp. 537–559, 1998.

[71] A. Evans, *Advances in Nonlinear Signal and Image Processing*, ch. 12, pp. 329–356. EURASIP Book Series on Signal Processing and Communications, Hindawi Publishing Corporation, 2006.

[72] P. Trahanias and A. Venetsanopoulos, "Vector order statistics operators as color edge detectors," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, pp. 135 –143, Feb. 1996.

[73] J. M. Prewitt, "Object enhancement and extraction," *Picture processing and Psychopictorics*, vol. 10, no. 1, pp. 15–19, 1970.

[74] I. Sobel and G. Feldman, "A 3x3 isotropic gradient operator for image processing," 1968.

[75] D. Marr and E. Hildreth, "Theory of edge detection," Tech. Rep. AIM-518, MIT Artificial Intelligence Laboratory, Apr. 6 1979.

[76] F. Samopa and A. Asano, "Hybrid image thresholding method using edge detection," *International Journal of Computer Science and Network Security*, vol. 9, no. 4, pp. 292 – 299, 2009.

[77] L. Lam, S. Lee, and C. Suen, "Thinning methodologies-a comprehensive survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, pp. 869–885, 1992.

[78] R. Duraiswami and R. Duraiswami, "Canny edge detection on nvidia cuda," *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, 2008.

[79] J. Fung, "Computer vision on the gpu," *GPU Gems*, vol. 2, pp. 649–666, 2005.

[80] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *2010 First International Conference on Networking and Computing*, pp. 279–280, IEEE, 2010.

[81] Z. Guo and R. W. Hall, "Parallel thinning with two-subiteration algorithms," *Commun. ACM*, vol. 32, pp. 359–373, Mar. 1989.

[82] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Commun. ACM*, vol. 27, pp. 236–239, Mar. 1984.

[83] NVIDIA, "NVIDIA CUDA SDK," March 2015.
`http://developer.nvidia.com/cuda-downloads`
(last accessed 02/04/2015).

[84] V. Podlozhnyuk, "Image convolution with cuda," *NVIDIA Corporation white paper, June*, vol. 2097, no. 3, 2007.

[85] J. Stam, "Convolution soup," 2009.
`http://www.nvidia.com/content/GTC/documents/1412_GTC09.pdf`
(last accessed 06/04/2015).

[86] NVIDIA, "CUDA Samples."
`http://docs.nvidia.com/cuda/cuda-samples/`
(last accessed 09/03/2015).

[87] Prenhall, "ImageProcessingPlace.com."
`http://www.imageprocessingplace.com/root_files_V3/image_databases.htm`
(last accessed 01/02/2015).

[88] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel genetic algorithm on the cuda architecture," in *Applications of Evolutionary Computation*, pp. 442–451, Springer, 2010.

[89] P. Pospíchal and J. Jaros, "Gpu-based acceleration of the genetic algorithm," *GECCO competition*, 2009.

[90] U. of Waterloo, "Mona Lisa TSP Challenge."
`http://www.math.uwaterloo.ca/tsp/data/ml/monalisa.html`
(last accessed 15/07/2015).

[91] T. Stützle and H. H. Hoos, "MAX-MIN ant system," *Future Gener. Comput. Syst.*, vol. 16, pp. 889–914, Jun. 2000.

[92] T. Stützle, "Parallelization strategies for ant colony optimization," in *Fifth Int. Conf. on Parallel Problem Solving from Nature (PPSN-V)*, pp. 722–731, Springer-Verlag, 1998.

[93] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," in *Fifth Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS)* (M. Dorigo, L. M. Gambardella, M. Birattari, A. Martinoli, R. Poli, and T. Stützle, eds.), vol. 4150 of *Lecture Notes in Computer Science*, pp. 224–234, Springer Verlag, 2006.

[94] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to parallel computation: P-completeness theory*, vol. 200. Oxford university press Oxford, 1995.