A Dynamic Programming Model To Solve Optimisation Problems Using GPUs

Jonathan Francis O'Connell

School of Computer Science & Informatics

Cardiff University

A thesis submitted in partial fulfilment

of the requirement for the degree of

Doctor of Philosophy

College of Physical Science and

Engineering

January 2017

For my wife Emma,

who made all this possible.

Declaration

Acknowledgements

Throughout the course of this PhD I've had the support of many people, all of which deserve an honourable mention here. Firstly, I must thank Christine Mumford, my long-suffering supervisor, whose invaluable support and advice have kept my research on track these last four or so years. I must mention the Cardiff University COMSC department for providing a superb working environment, a support network when work got tough, and a distraction during long days. Special thanks are due to Matt John who has been an office mate, flat mate, and good friend over the last few years. I would like to thank my supervisor at Bristol University, Elisabeth Oswald, for being so understanding and allowing me time to work on my thesis during my employment there as an RA. I must also thank my friends for keeping me sane during this process: Mads, Isaac, Rob, George, and James providing welcome relief from my work. The girls of Number 66, Abbi and Elin, deserve a special mention for enduring me far too many days a week, and providing me with unending hot beverages and food. The future in-laws, Mark and Valeria, for giving me a comfy sofa and home cooked dinners, and all the Findlay-Wilsons for looking after me so well. Finally, of course, I must thank my family – my Mum and Dad for providing me, often much-needed, emotional and financial support across the years; my brothers Jason, for giving me a cosy change of scenery every so often, and Christian, whose email to the head of school got me funding in my first year. To this day I have no idea what he said... Most of all, I must thank Emma. Without her encouragement, and steadfast belief in me, I would have never reached this point.

Without you all, this would have been impossible. Thank you.

Abstract

This thesis presents a parallel, dynamic programming based model which is deployed on the GPU of a system to accelerate the solving of optimisation problems. This is achieved by simultaneously running GPU based computations, and memory transactions, allowing computation to never pause, and overcoming the memory constraints of solving large problem instances. Due to this some optimisation problems, which are currently not solved in an exact manner for real world sized instances due to their complexity, are moved into the solvable realm. The model is implemented to solve, a range of different test problems, where artificially constructed test data is used to ensure good performance even in the worst cases. Through this extensive testing, we can be confident the model will perform well when used to solve real world test cases. Testing of the model was carried out using a range of different implementation parameters in relation to deployment on the GPU, in order to identify both optimal implementation parameters, and how the model will operate when running on different systems. All problems, when implemented in parallel using the model, show run-time improvements compared to the sequential implementations, in some instances up to hundreds of times faster, but more importantly also show high efficiency metrics for the utilisation of GPU resources. Throughout testing emphasis has been placed on GPU based metrics to ensure the wider generic applicability of the model. Finally, the parallel model allows for new problems to be defined through the use of a simple file format, enabling wider usage of the model.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Overview

This thesis presents a generic parallel model that seeks to enable a subset of optimisation problems to be solved, both quickly and efficiently, through the use of Graphic Processing Unit (GPU) based techniques. A parallel model is a method of mapping algorithms and data to multiple processors by splitting it into multiple parts, such that all parts can be executed simultaneously in an effort to reduce computation time. A GPU is a highly parallel processing unit, originally designed to render 3D images, but more recently can be utilised for general purpose programming. Performance metrics have been taken using specialised GPU profiling tools to validate the efficiency of the proposed model. Also described is a file format which seeks to allow the implementation of optimisation problems on the GPU, with minimal programming knowledge from the end user. Analysis and discussion of the observed metrics allows conclusions to be drawn on the suitability and applicability of the model in different scenarios.

In this chapter a high level overview of the work detailed in the thesis is presented, as well as the contributions stemming from this. Also in this chapter is an outline of the thesis structure.

1.1 Introduction

Solving discrete optimisation problems exactly is desirable, as, by definition, this is guaranteed to provide the optimal solution. This thesis is focused on the field of solving optimisation problems through the use of general purpose graphics processing unit (GPGPU) programming. “Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth” [71]. GPGPU programming is a relatively new field of programming, which allows end users to program GPUs for uses other than the purpose of rendering 2D images of 3D spaces: “graphics processors (GPUs) become attractive because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and general purpose instruction sets, including support for both single-precision and double-precision [...] arithmetic” [17]. GPUs in many applications can prove to be considerably faster than their CPU counterparts; “the reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialised for compute-intensive, highly parallel computation [...] and is therefore designed such that more transistors are devoted to data processing rather than data caching and flow control” [84]. This has led to the rise of languages to support the implementation of programs on the GPU such as the proprietary, closed source, NVIDIA CUDA [70] - NVIDIA Compute Unified Device Architecture, which allows programs to be deployed on NVIDIA hardware, and the focus of this thesis is limited to CUDA. However, the unique architecture of the GPU presents unique programming challenges [31, 98], and developing parallel models to efficiently use the GPU is important: “... should redirect efforts in GPGPU research from ad-hoc porting of applications to establishing principles and strategies that allow efficient mapping of computation to graphics hardware” [82].

More specifically, this thesis considers using the extensive computational resources of the GPU to accelerate the solving of classic, hard problems with high computational complexities, through the use of a technique called dynamic programming [19]. These common problems form the core of many real world algorithms, from vehicle routing to cryptography, and any performance gains should be widely beneficial. The solving methods considered here are exclusively exact methods, meaning this work not only improves run time but also guarantees solution quality is not only maintained, but is in fact maximised, and possibly improved over inexact methods. This is done with the motivation that GPUs have such a high number of cores, exact solving may be possible for hard problems.

As aforementioned [82], much of the previous work relating to the development of dynamic programming algorithms for deployment on the GPU has been targeted specifically at individual problems, rather than looking at overarching parallel methods. Considering only some of the test problems described within this thesis, a range of algorithms specifically targeting each is available [14, 99, 41]. We present instead a single, generic model that is suitable for a range of problems, even beyond the ones for which it was originally designed. As such, the study of a model for a wide range of uses such as this, is a novel direction of research. The work presented in this thesis contributes further knowledge to the discussion on developing suitable parallel models that target GPGPU hardware for a wide range of problems.

To allow the implementation of a range of different problems we present a basic syntax which defines the problem function, input data and type, and some implementation parameters, allowing an end user to implement further problems easily. All test problems in this thesis, which are defined as suitable for parallelism through the model, are implemented through this syntax, demonstrating its validity.

Changing how the data maps to the physical hardware of GPU, through parameters such as the block size, can have an effect on both run-time and resource utilisation. This thesis also examines the effect the choice of mapping strategy has, and what factors influence this, such as: the specific problem; the class of problem; or whether selecting the optimal strategy is problem agnostic and merely a factor of our model. Correctly mapping data to the hardware is found to have a significant impact on the performance of the parallelism, with slight differences from problem to problem, and this is discussed in detail in the performance results.

The thesis provides an analysis of the model in terms of efficiency, using NVIDA CUDA based metrics, meaning not only the run time is considered – it should be assumed a device with hundreds of cores will improve the run-time – but also how effectively the available underlying resources of the GPU are being used. This analysis gives insight to the performance of the model, not restricted to the test problems examined in this thesis, allowing predictions to be made as to how it will perform more generally for different problems and on future iterations of GPU hardware.

1.2 Thesis Aims

The main aim of this thesis is to present an abstract and generic solving methodology that enables the mapping of optimisation problems, which can be solved through dynamic programming techniques, onto the graphics processing unit with the goal of improving the run-time of these computationally challenging problems. It aims to provide an investigation into how to effectively solve these problems using the GPU, considering other factors beyond run-time such as memory efficiency and other overheads inherent in GPU programming. Through this, we aim to increase the knowledge related to running this class of problem on a GPU.

1.3 Contributions

The main contributions offered by this thesis are:

- A generic parallel model which runs on the GPU that, through the use of dynamic programming techniques, allows a subset of optimisation problems to be solved by effectively using the high performance computing facilities offered by the GPU architecture. Problems which can be solved include the the longest common subsequence problem, the travelling salesman problem, and the knapsack problem. The exact range of problems that can be solved is dictated by the dependency structure of the specific problem - this is discussed in detail in Section 3. The model is analysed and profiled, and performance figures are provided to demonstrate the effectiveness of the model.

- An efficient memory structure and memory management, that seeks to reduce the memory complexity on the GPU of the given input problems, allowing larger scale problem instances to be solved. This is an especially important consideration when running code on the GPU, as the amount of available memory is often more restrictive than normal programming environments, meaning it quickly limits the feasible problem size. As well as being memory efficient, it is required the model achieves this whilst simultaneously not causing excessive additional computational overhead. Again, profiling is used to validate that it achieves both goals.

- A generic file format based on Backus-Naur Form (BNF), that allows different problem definitions into be quickly input to the program. The file contains basic information about the problem to be solved, such as input data types, as well as information about the amount of memory the problem requires. This is then coupled with a very small function definition which defines the logic of the problem, and the mapping of the dynamic programming definition onto the input data, meaning that the model remains generic - allowing new problems to be added quickly and easily.

1.4 Thesis Structure

This thesis is laid out as follows:

- Beginning in Chapter 2, the background work that underpins this thesis is presented, and relevant problems and literature are introduced. In Section 2.1 the test problems that have been selected are formally defined. In addition to a mathematical definition, both the computational and memory complexity required to solve them using naive approaches are given. Section 2.2 formally introduces and defines what is meant by the term *dynamic programming*, and provides the criteria that a problem must satisfy for dynamic programming to be applicable. Next we show how dynamic programming can be applied to solve the introduced test problems, and present some basic solving algorithms. Moving into Section 2.3, some alternate solving methodologies compared to dynamic programming are very briefly introduced, to help illustrate in which settings dynamic programming is appropriate. Finally, in Section 2.4, the concept of parallel programming is introduced and defined, as well as the benefits and drawbacks associated with it. The different paradigms of parallel programming are formally defined, and the GPU programming language we use, NVIDIA CUDA, is introduced and explained.

- Chapter 3 presents the model this thesis is proposing. Starting in Section 3.1, a high level design of the model is presented, demonstrating how it maps data to the physical resources of the GPU, and how the memory is managed when the model is in use. Next, in Section 3.2, the specifics of the implementation are considered, and small scale benchmarks are provided to justify the design decisions taken. This section also discusses in detail the file format used to allow new problems to be defined. The chapter concludes with Section 3.3, where we compare our model to the existing algorithms available in the literature, and identify the main contributions our model offers.

- Next, Chapter 4 details how to implement the introduced problems using our parallel model. Section 4.2 goes on to introduce potential optimisations that can be made, without changing the overarching paradigm. In Section 4.3 we discuss which problems were found to be unsuitable for implementation through the model, or required it to be changed considerably, and attempt to classify the subset of problems that is suitable.

- Chapter 5 introduces the hardware that will be used for the analysis of the model, as well as the profiling and performance metrics that will be recorded during testing. Additionally, detail is given on how the test instances of each problem are generated. Also in this chapter, some reference benchmarks for the hardware used is provided to give an indication as to baseline and ideal performance.

- Moving into Chapter 6, we present the main performance results from the analysis of the model, presenting results detailing run-time, and the recorded CUDA metrics. We seek to generalise these results, in terms of the number of GPU blocks, and GPU block sizes, allowing the results to be extrapolated and predictions made as to the performance when running on alternate GPU hardware.

- Finally, Chapter 7 summarises the work, and draws conclusions based on the results that have been outlined in this thesis. In Section 7.6, some ideas and theories are presented as to the direction in which this work could be taken next.

1.5 Publications

1.5.1 Model Design

The work presented in Chapter 3 covers the design of the generic parallel model, as well as how specific problems can be mapped onto the GPU using it.

An earlier version of this has been published in [74]:

- O'Connell, J. F. & Mumford, C. L. (2014), An Exact Dynamic Programming Based Method to Solve Optimisation Problems Using GPUs, 347-353. In Proceedings of the Second International Symposium on Computing and Networking, IEEE. **doi:10.1109/CANDAR.2014.27**

The performance of our parallel model is considered in terms of speed and computational efficiency when applied to instances of the knapsack problem, the longest common subsequence problem, and the travelling salesman problem.

In this paper we found our model performed well with all test problems, achieving a considerable level of speed up factor compared to the classic CPU implementation, as well as utilising the available resources of the GPU effectively. However, the paper posed further questions about improving the divergence of the code in order to improve the overall efficiency.

1.5.2 Implementation

In Section 4.3.2 the implementation of the all pairs, shortest path problem on the GPU is described. The algorithm detailed here has been used to support the work in the following submitted journal article:

- John, M. P., Mumford, C. L, Lewis, R, O'Connell, J. F. Exploring design issues for solving the urban transit routing problem using a multi-objective evolutionary algorithm. In IEEE Transactions on Evolutionary Computation. *Under review.*

A parallel implementation of the all pairs shortest path problem is used to support the implementation of an algorithm which uses meta-heuristic techniques, in an effort to design efficient bus route networks based on an underlying road network.

Whilst the focus of the article was not that of the performance of the GPU implementation, it does discuss the fact that the GPU implementation of the all pairs shortest path problem produces run times several hundred times better than that of the classic serial CPU counterpart. The study also notes that as the problem size increases, the speedup factor offered by the GPU also increases, until it becomes steady near the 800% mark.

Summary

In this chapter the thesis topic was introduced, as well as the motivation for undertaking the work detailed within. The key contributions offered were detailed, and an overall structure of the following document was presented. The next chapter will introduce the background concepts upon which the work in this thesis builds.

Chapter 2

Background

Overview

This chapter introduces background topics and problems, as well as existing work using NVIDIA CUDA 6.5 [52], that is relevant to the research in this thesis.

Beginning in Section 2.1, the problems which this thesis is concerned with solving are introduced and defined: namely, the longest common subsequence problem and the edit distance problem, the knapsack problem, the all pairs shortest path problem and the Manhattan tourist problem. Both graphical representations and mathematical definitions of these are provided.

Moving into Section 2.2, the method of programming and solving a problem, called *dynamic programming* [9], is introduced. The benefits and drawbacks of using such an approach to solve a problem are considered. The solution methodologies based on dynamic programming techniques for the introduced problems are defined.

Next, in Section 2.3, other alternative classical methods of solving the aforementioned problems are considered and discussed. We address *inexact* solving methodologies, heuristic and meta-heuristic methods, as well as *exact* methods.

Finally, in Section 2.4, parallel programming is introduced. Discussed are the concepts of parallel programming, the different paradigms, as well as the common use cases and issues associated with it. Next we consider how effectively a dynamic programming based algorithm can be mapped onto a parallel processing model. Lastly, the programming model offered by NVIDIA CUDA is outlined, before finally discussing existing algorithms and models that employ the CUDA programming environment in order to solve the introduced problems.

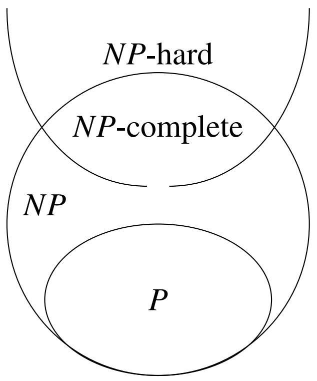


Fig. 2.1 How the different complexity classes interact and overlap in terms of *P* and *NP*

2.1 Problem Introduction

Introduced in this section are the problems we are seeking to solve with the parallel model presented by this thesis. Whilst all of these are easily solved for limited size instances, due to the high computational complexity they very quickly become in-feasible to solve exactly as the size of the problem grows. It is therefore very common to solve these inexactly using methods such as heuristics. However, as parallel programming seeks to reduce the run time of executing a program, these problems stand to benefit from being implemented in parallel as this will allow larger instances to be solved exactly.

2.1.1 NP Complexity

All problems which will be introduced fall under the complexity class of *NP*, therefore we first define this. The '*P*' in *NP* is defined as the computational complexity when a decision problem can be solved by a deterministic Turing machine, requiring a polynomial amount of computation proportional to the size of the input. Corbham's thesis states that, generally, a problem can be solved feasibly on a computation device if the problem lies within the *P* class [40]. Obviously this is still highly dependent on the size of the input, but is a good starting point when considering the feasibility of solving a problem.

The computation class *NP* refers to non-deterministic polynomial time problems. These are the class of decision problems where correct solutions to the problem can be accepted by a non-deterministic Turing machine in polynomial time. The complexity class *P* is contained within the larger class of *NP*.

NP-complete problems are problems which are enclosed in both the NP and NP-hard complexity classes. Although solutions to these problem can be verified in polynomial time, there is no quick way of generating the solution in the first place. Formally, this is defined as: a problem *p* in the class *NP* is *NP*-complete, if all other problems in *NP* can be transformed to *p* in polynomial time. An example of reducing an *NP*-complete problem to different *NP*-complete problems is reducing the travelling salesman problem to the Hamiltonian cycle problem, which in turn can be reduced again to the vertex cover problem. Generally, *NP*-complete problems are considered more difficult to solve than *NP*, because if there is a method of quickly solving an *NP*-complete problem, there is a quick method of solving all *NP* problems (as every problem in *NP* can be reduced to an *NP*-complete problem). As there is no computationally quick way of generating solutions to these problems, they are commonly solved through the use of approximation algorithms, or heuristic methods. These methods generate solutions to the problem, albeit ones that may not be optimal. The relationship between the classes is shown in Fig. 2.1.

When discussing *NP*-complete problems, it is a common misconception that the *NP* stands for *non polynomial* time, referring to the fact there is no know polynomial time algorithm to solve the problem. Whilst this is likely true, as these problems are so difficult to solve, this has never been proven.

NP-hard problems are problems which are at least as hard as the hardest problem within the *NP* class. However, they are not required to be decision problems, nor in the *NP* class at all. Formally, a problem *p* is *NP*-hard if there is an *NP*-complete problem *y* which can be reduced in polynomial time to *p*. As all *NP*-complete problems can be reduced to any

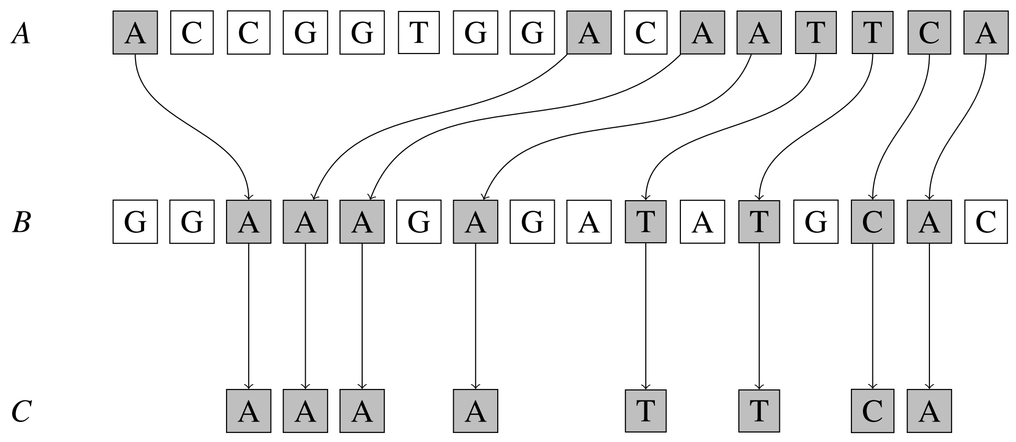

A
A C C G G T G G A C A A T T C A
B
G G A A A G A G A T A T G C A C
C
A A A A T T C A

Fig. 2.2 The longest common subsequence (C) being extracted from two input strings (A and B)

other NP-complete problem in polynomial time, all NP-complete problems can be reduced to NP-hard problems. An example of an NP-hard problem is the halting problem - given a program, and an input, will it halt? This is a decision problem but it is not in NP, yet it is clear that any NP-complete problem can be reduced to this.

2.1.2 The Longest Common Subseqence Problem

> "String comparison is a central operation in various environments: a spelling error correction program tries to find the dictionary entry which resembles most a given word, in molecular biology we want to compare two DNA or protein sequences to learn how homologous they are, in a file archive we want to store several versions of a source program compactly by storing only the original version" [11]

The longest common subsequence (LCS) problem has the goal of finding the longest subsequence that is common to a given set of input sequences. This input set is commonly assumed to consist of two sequences, but there is no limit and n sequences can be analysed, although each additional sequence increases the computational complexity.

The problem is well defined by Hirschberg [43]. String $C = c_1, c_2, \ldots, c_i, \ldots, c_p$ is a subsequence of string $A = a_1, a_2, \ldots, a_i, \ldots, a_m$ if there is a mapping $F : \{1, 2, \ldots, p\} \rightarrow \{1, 2, \ldots, k\}$ such that $F(i) = k$ iff $c_i = a_k$ and F is a strictly increasing function, where n is the length of the first string A, m is the length of the second string B, and p is the length of the subsequence C. C can be formed by deleting $m - p$ (not necessarily adjacent) symbols from A.

From this a common subsequence is defined as: C is a common subsequence of A and $B = b_1, b_2, \ldots, b_i, \ldots, b_n$ iff C is a subsequence of A and a subsequence of B. Therefore C is the longest common subsequence of A and B iff:

- C is a common subsequence of A and B
- There is no common subsequence D, for which the length of D is larger than C

A graphical example of the longest common subsequence problem can be found in Fig. 2.2.

For the general case of an arbitrary number of sequences, the problem is known to be NP-hard. To solve this through a simple naïve approach [60], assuming an input of N sequences, where the length of each sequence is defined as $s_1, s_2, \ldots, s_i, \ldots, s_N$, it would be a case of testing each of the 2^{s_1} subsequences of the first input sequence to identify if they are subsequences of the remaining input sequences. These subsequences are checked in linear time for the remaining input sequences, which leads to the complexity of $O\left(2^{s_1} \sum\limits_{i>1} s_i\right)$. Obviously this complexity is far too high to be practical for any reasonably sized input data.

The problem is of interest as it has many uses and provides the underpinning for a wide variety of different algorithms. The most common of these uses is that it is very prevalent in bio-informatics algorithms, as the structure of searching for subsequences is very similar to matching patterns within DNA and protein data [3, 76]. Here a parallel implementation could speed things up considerably, allowing longer strings to be analysed, or more strings to be analysed simultaneously.

This problem is also at the core of other algorithms such as compression algorithms or version control systems, as through the identification of subsequences it allows the difference between files, and lines of files, to be quickly identified [4]. Here a parallel version would simply reduce the execution time of such tools and utilities.

Due to the problems widespread uses, it has been the focus of much study, and as such there are many polynomial time algorithms available for when the size of the set of input strings is restricted, or when the alphabet available to the input strings is constrained. The most simplistic and widespread of these is a simple dynamic programming approach [43], which allows the solving of two string problems in polynomial time based on the length of the longest input string. These are discussed in detail in Sec. 2.2. Using this method compared to the brute-force approach reduces the computational complexity to $O(mn)$, when only two input sequences are being processed, or $O\left(N\prod_{i=1}^{N}s_i\right)$ for the general case where N is the number of input sequences, and s the length of each sequence.

Edit Distance

The edit distance problem is a small extension to the longest common subsequence problem that seeks to apply metrics to the difference between two strings, and to apply a *cost*, or *distance*, that it would take to transform one string into another.

There are several methods to define the metrics used in the edit distance problem, and we chose to adopt a set of rules for our sample problems, called the Levenshtein distance [50]. The rules it defines for transforming a string are (where λ is the empty string):

- *Insertion* of a symbol. Inserting character c into string ab produces acb, or $\lambda \rightarrow c$.
- *Deletion* of a symbol. Deleting c from acb produces ab, or $c \rightarrow \lambda$.
- *Substitution* of a symbol. Substituting c for d in acb produces adb, or $c \rightarrow d$.

$$\begin{tikzcd}
\textbf{I}\text{NTENTION} \arrow[d, "\textit{(delete I)}"] \\
(\lambda)\textbf{N}\text{TENTION} \arrow[d, "\textit{(substitute N for E)}"] \\
\text{E}\textbf{T}\text{ENTION} \arrow[d, "\textit{(substitute T for X)}"] \\
\text{EXE}(\lambda)\text{NTION} \arrow[d, "\textit{(insert C)}"] \\
\text{EXEC}\textbf{N}\text{TION} \arrow[d, "\textit{(substitute N for U)}"] \\
\text{EXECUTION}
\end{tikzcd}$$

Fig. 2.3 The edit distance moving from the string *INTENTION* to *EXECUTION*, which has a Levenshtein distance of 8

Any arbitrary values can be applied to these scores; however, throughout this thesis we will assume that deletions and insertions have a cost of 1, and substitutions a cost of 2. Considering the example problem in Fig. 2.3, the presented string transformation would have an edit distance of 8.

As with the longest common subsequence, the algorithm has implications in bio-informatics processing, where the *optimal alignment* of two strings can be defined as the alignment of the two strings with the smallest edit distance [43].

The algorithmic implementation of this is very similar to the dynamic programming approach adopted to solve the longest common subsequence problem [94]. However, we also consider it as a test problem in this thesis as it demonstrates the usefulness of our model in solving different problems, when the underlying algorithms are altered to serve different purposes.

The benefits of parallelising the algorithm to solve this problem are very similar to that of parallelising the longest common subsequence problem: longer strings can be computed in a given time frame without the requirement of being separated into multiple smaller strings, as well as accelerating the computation of already feasible strings.

2.1.3 Knapsack Problem

> "A well-known combinatorial problem that finds applications to capital budgeting problems, loading problems, and solutions of large optimization problems is the knapsack problem." [45]

The Knapsack Problem (KSP) is a combinatorial optimisation problem concerned with: given a set of items with an associated mass and profit value for each, select the subset of items such that a given capacity constraint is not violated by the cumulative mass of the items, whilst simultaneously maximising the cumulative profit of all the items selected.

There are many variants of the knapsack problem, but one of the most common is the 0-1 knapsack problem, which restricts the amount of times an item is selected to either zero or one. In this sense, it follows the real world analogy of an item being either selected and placed in the knapsack, or it being excluded from the knapsack and left out.

This can be defined thus [78]: let there be a set of items, $z_1, z_2, \ldots, z_i, \ldots, z_n$ where z_i has a value $v_i \in \mathbb{N}_+^*$ and a weight $w_i \in \mathbb{N}_+^*$. x_i is a boolean defining whether the item z_i is selected. The maximum weight of the selected items cannot exceed $W \in \mathbb{N}_+^*$.

$$\begin{aligned} &\max \sum_{i=1}^{n} v_i x_i \\ &\text{subject to: } \sum_{i=1}^{n} w_i x_i \leq W, \qquad x_i \in \{0,1\} \end{aligned} \tag{2.1}$$

This maximises the sum of the value of the selected items, whilst ensuring that the weight constraint, W, is not violated. x_i denotes how many times item i has been selected, and by altering the constraint applied to x_i, the problem can also be changed as required to reflect the real-world problem that is being mapped to it. For example, allowing x to be any value such that the constraint becomes $x_i \geq 0$, this allows as many copies of an item as required to be selected, and forms the *unbounded knapsack problem.* Similarly, if there is a different amount of each item available, the x constraint can be changed to $x \in 0, 1, \ldots, c_i$, where the amount of copies of each item (x_i) is restricted to a given limit, c_i. This is the *bounded knapsack problem.* An example instance of the knapsack problem is shown in Fig. 2.4, where the capacity of the knapsack is 18, and the maximum profit attainable from the item set is 49.

Due to the lack of a polynomial time algorithm that can assert for all cases whether or not a given solution to the problem is optimal, in terms of complexity the knapsack problem is *NP*-hard [78]. However, a reduction of the knapsack problem, referred to as the *decision*

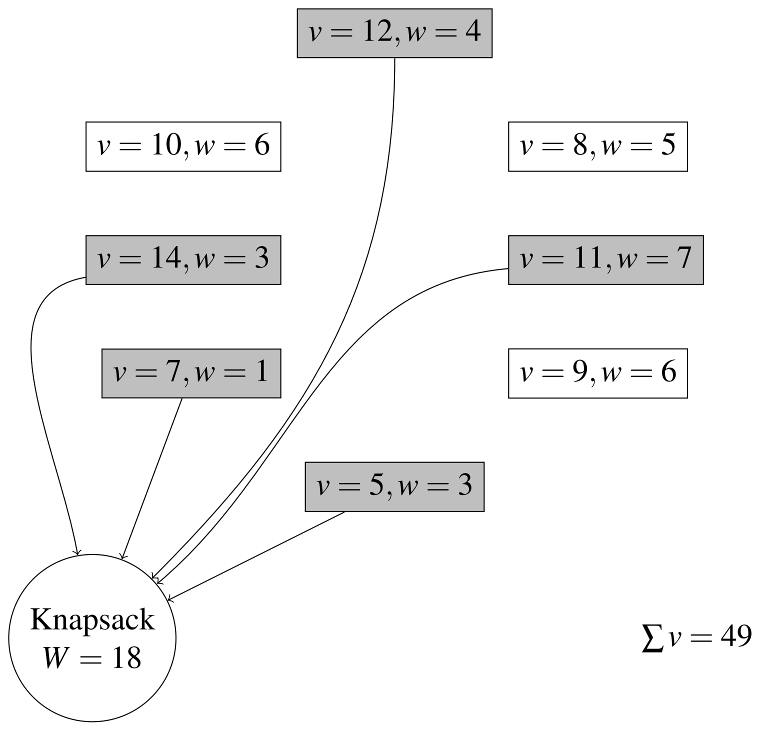


Fig. 2.4 A knapsack instance where the subset of optimal items from a given set of input items is highlighted

problem, seeks to bound the problem and define whether or not a given profit value can be attained without exceeding the weight constraint, and this is NP-complete.

There exists dynamic programming algorithms that can solve the problem in pseudo-polynomial time, which will be the focus of this thesis and are introduced in Sec. 2.2. This dynamic programming algorithm runs in $O(nW)$ time. Furthermore there has been a lot of focus on approximation methods - methods which seek to find a subset of solutions based on a reduced scaled version of the inputs. in an effort to then extrapolate to the real solution set. For the knapsack problem there are many approximation schemes which run in fully polynomial time. Finally, as with most optimisation problems, there exist many algorithms which seek to solve the problem inexactly through heuristic and meta-heuristic methods, which are briefly discussed in Sec. 2.3.1.

As the knapsack problem can be effectively represented as a simple linear integer program [45], higher level problems can be reduced to the knapsack problem, and it provides the underpinnings of many decision and optimisation problems. Some of the most common uses are in work-flow scheduling or manufacturing, where an employee or a piece of machinery can be represented as a knapsack; in this instance, variations of the knapsack problem where there are multiple copies of an item become beneficial. It is also often used as a tool to analyse likely investments, where each has a specified return (the value of the knapsack item), but an initial cost (the weight of the knapsack item). The hardness of bigger instances of the problem also means that it can lend itself to supporting cryptographic systems [62].

2.1.4 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) [7] is a classic NP-hard computer science problem. The goal of the problem is thus: given a set of vertices, construct a set of edges which connects all these vertices in an acyclic manner, where the cumulative length of the edges is sought to be minimised. The typical analogy given is a salesman travelling between a set of cities, who seeks to minimise the total driving distance.

This is formally defined, as given a set of vertices, which are cities, these are labelled $1,2,...,n$. As aforementioned, each one of these vertices must be visited once, and for it to be a valid tour, the route must start and end at the same point. The naive, brute force algorithm has a huge computational complexity of $O(n!)$, making exact solving impossible for all but the smallest instances; however, it is at the core of many vehicle routing and scheduling algorithms. To represent the construction of the route, let us define a variable x where for an n city problem x_{ij} is defined as:

$$x_{ij} = \begin{cases} 1 & \text{the route goes from city } i \text{ to city } j \\ 0 & \text{o.w.} \end{cases}$$

Let the cost of moving between two cities i and j be defined as $c_{i,j}$. Then with the use of a temporary variable, u_i Miller [63] shows how this problem can be defined as an integer linear program:

$$\min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij} \tag{2.2}$$

$$\text{subject to: } 0 \leq x_{ij} \in \{0,1\} \leq 1 \qquad i,j = 0,...,n \tag{2.3}$$

$$u_i \in \mathbb{Z} \qquad i = 0,...,n \tag{2.4}$$

$$\sum_{i=0, i \neq j}^{n} x_{ij} = 1 \qquad j = 0,...,n \tag{2.5}$$

$$\sum_{j=0, j \neq i}^{n} x_{ij} = 1 \qquad i = 0,...,n \tag{2.6}$$

$$u_i - u_j + n x_{ij} \leq n - 1 \qquad 1 \leq i \neq j \leq n \tag{2.7}$$

Constraints 2.3 and 2.4 ensure that each city on the tour can only be arrived at from exactly one other city. Equalities 2.5 and 2.6 requires that from each city on the tour, there is a departure to exactly one other city. Finally, constraint 2.7 ensures that there is only a single tour covering all cities, and that there cannot be multiple, simultaneous disjointed tours.

There exists a dynamic programming based algorithm to solve this, called the Held and Karp algorithm, which will be discussed in Sec. 2.2; it reduces the computational complexity to $O\left(2^n n^2\right)$, with an associated memory complexity of $O\left(2^n\right)$. However, this is obviously still too high for most real world sized instances to be solved in a reasonable amount of time. Due to this, the problem is very rarely solved using exact methods and therefore, this problem is solved almost exclusively through inexact methods such as heuristics. A brief introduction to inexact methods is provided in in Sec. 2.3.1.

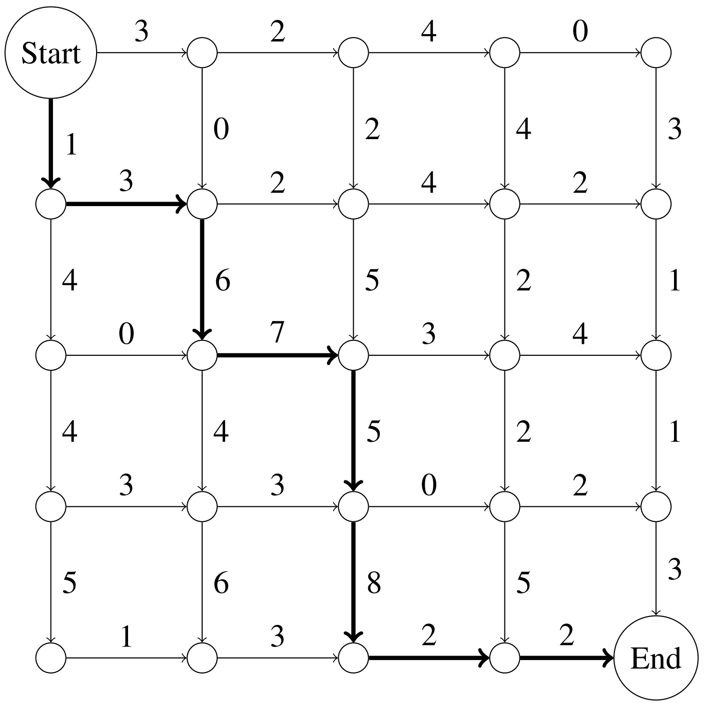


Fig. 2.5 A Manhattan tourist problem instance. The number of landmarks on each street denoted as the weight of the edges, the optimal solution is highlighted in the figure

2.1.5 Manhattan Tourist Problem

The aim of the Manhattan toursit problem is to find a route that crosses the borough of Manhattan, such that the amount of landmarks visited on the route is maximised. More generally each edge is assigned a weight, proportional to the number of landmarks on the edge, and the goal is to travel from a start vertex to a goal vertex maximising the total weight of the edges traversed. An example of this is shown in Fig. 2.5.

Within the Manhattan tourist problem the possible moves at each vertex is *south* or *east* to prevent cycles forming in the graph. Therefore it should be apparent that this problem is similar to the travelling salesman problem, with a different set of constraints; the goal is to traverse from A to B, whilst maximising the edge weight (rather than minimising distance in the case of the TSP), and there is no constraint to visit each node. Unlike the travelling salesman problem, this problem is not NP-hard.

This generalised *toy* problem can form the basis of more advanced problems; it shares the same foundation as the longest common subsequence problem, or the edit distance problem, whilst also being similar in its use case to the travelling salesman problem. Parallelising this problem is therefore beneficial, as demonstrating a model that can solve this simplistic case can have implications in solving more specific problems, and can likely also be used to solve other problems with minimal adaptation.

A naïve recursive algorithm to solve this problem would run in exponential time. Therefore there are dynamic programming solutions available to this problem [49], which are similar to those available for both the longest common subsequence problem [43] as well as the edit distance problem; again these will be discussed in Sec. 2.2. The complexity of such algorithms is dictated by the x and y dimensions of the input grid being considered, and therefore the computational complexity of the algorithm is $O(xy)$.

2.1.6 All Pairs, Shortest Path Problem

The all pairs shortest path problem is a generalisation of the previously introduced Manhattan Distance Problem. It is concerned with finding the shortest routes through a directed acyclic graph using every combination of pairs of nodes in the graph as a start and end vertex.

The shortest path problem is defined as:

$$\min \sum_{ij \in A} w_{ij} x_{ij} \tag{2.8}$$

subject to: $x \geq 0$

$$\forall i \sum_j x_{ij} - \sum_j x_{ji} = \begin{cases} 1 & i = s \\ -1 & i = t \\ 0 & o.w. \end{cases}$$

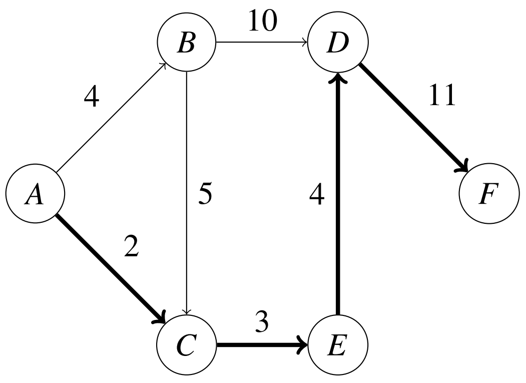


Fig. 2.6 A directed graph with the cost shown for traversing each edge, highlighting the shortest path from A to F.

A graphical representation of a small example of the shortest path problem is presented in Fig. 2.6.

In this notation, s is representing the source node, t the target node and w_{ij} is the cost of traversing the edge from i to j. x_{ij} is used as an indicator to denote whether or not the edge is present in the shortest path. From this definition, the problem moves to an all pairs, shortest path problem working out the shortest path between s and t when s and t are set to all pairs of nodes in the graph.

This is represented as an adjacency matrix, which is a matrix of size $v \cdot v$ where each cell (i, j) of the matrix denotes the edge weight between node i and j, or ∞ if there is no feasible path. In the case of the all pairs shortest path problem, once the algorithm has finished executing (i.e. the problem has been solved), the values in the adjacency matrix will be updated to reflect the shortest path between each i and j.

A brute force approach to solving this problem would obviously require considering every single edge between all the vertices of the graph, which would lead to the time complexity of $O\left(v^2\right)$. However, the method we seek to parallelise to solve this problem [29] is introduced in detail in Sec. 2.2.3.

As with the previous vehicle routing based problems, paralellisation is desirable as it allows larger matrix sizes to be computed, which may mean that entire route network can be calculated at once rather than having to break them down into smaller sub networks leading to efficiency savings.

2.2 Dynamic Programming

In this section the method of programming called *dynamic programming* is introduced, as it forms the use of dynamic programming based techniques the parallel model proposed in thesis is developed.

2.2.1 Definition

Dynamic programming allows large complex problems to be solved by breaking them down into smaller sub-problems, which are then solved independently, the results of which are combined to form the solution to the original problem [24].

The first property the problem must have is that it must be possible to break the problem down into *overlapping sub-problems* [22]. This means the results from the sub-problems are reused several times, or in a recursive situation the same problems are solved repeatedly, rather than creating a new problem every single time [19]. This is not to be confused with an approach such as divide and conquer, where the problem is broken down into to separate non-overlapping sub-problems which are then solved independently to reconstruct a solution.

The second property the problem is required to have is that it must have an optimal sub-structure. The best way to describe this is through the example of solving an optimisation problem starting from a time period t and ending at time period T. It should be apparent that it is required to solve sub problems s starting at later date first (subject to $t < s < T$). This is a very basic example of an optimal sub-structure. Another illustrative example would be to minimise the cost of a set of alternatives; where the search space can be partitioned into smaller subsets, and each of these alternatives only belongs to a single subset. These can be further reduced, until finding the minimal cost of each subset is trivial, and it is apparent that this also solves the original larger problem. This is an optimal sub-structure.

The concept of the optimal substructure in turn leads to the Bellman's Principle of Optimality [8] and the Bellman Equation [24], which is a necessary condition for optimality when using a dynamic programming based solving method. This equation shows that a dynamic programming algorithm to solve an optimisation problem can be solved through recursion by defining the relationship between the the *score* or *value* of the optimisation at one period of time, and the change in this at the next period of time. This relationship is called the Bellman Equation, which we now define.

Let a state of the problem at time t be called x_t, therefore the initial state of the problem will be called x_0. The set of possible actions at each time step, $a_t \in \Gamma(x_t)$, is dependent on the current state , where a_t is the set of variables a that can be altered at this specific time step t. The transition from a state x due to the variables of a we represent as $T(x_t, a_t)$, and the value or score of $T(x_t, a_t)$ we define as $F(x_t, a_t)$. Therefore the decision problem that requires optimisation can be defined as:

$$\begin{aligned} V(x_0) = \max_{\{a_t\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta F(x_t, a_t) \qquad (2.9) \\ \text{subject to: } a_t \in \Gamma(x_t) \\ x_{t+1} = T(x_t, a_t) \\ \forall t \in \{0, 1, \ldots, \infty\} \\ 0 < \beta < 1 \end{aligned}$$

In this notation $V(x_0)$ is the value functions, and this is dependent on the initial state as the overall solution to the problem is dependent on all states following the initial state.

Breaking this problem down into smaller sub problems, which is the basis of dynamic programming is addressed by Bellman as:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [8]

When a problem exhibits this structure, it means that is displays an optimal sub-structure, and as such is a suitable candidate for solving through dynamic programming. To mathematically represent the principle of optimality it is a case of considering the current time step separately to all future time steps, and then starting again for the next time step, as this new state will affect the whole future decision process. This can be represented as a recursive definition:

$$\begin{aligned} V(x_0) &= max_{a_0}\{F(x_0,a_0)+\beta V(x_1)\} \qquad (2.10)\\ \text{subject to: } & a_0 \in \Gamma(x_0)\\ & x_1 = T(x_0,a_0) \end{aligned}$$

which we can simplify further to:

$$V(x) = max_{a\in T(x)}\{F(x,a)+\beta V(T(x,a))\} \qquad (2.11)$$

This recursive relationship can be identified in all of the following dynamic programming solving methodologies of the introduced problems.

2.2.2 Dynamic Programming Methods

In computer science, implementing a dynamic programming based algorithm is either done through simple recursion to directly reflect the mathematical representation, or memoization which allows the partial solutions to be stored in a table or similar data structure.

In the case of large problem instances, most languages would very quickly hit a recursion limit, where the call stack becomes full and the function cannot recurse further. Therefore using memoization this limitation can be removed; by storing partial solutions in main system memory, future function calls simply lookup this value then continue the calculation. Memoization can also be referred to as tabling.

Tabling allows us to break a problem down into sub-problems and, when we solve a new sub-problem, consult the scoring table to see if it has already been solved. If it has, the stored solution is loaded, if it has not then the solution to the sub-problem is calculated and stored in the table. This saves computation, at the expense of storage. Finally, this scoring table can then be traversed to find the solution to the overall problem.

2.2.3 Solving Problems Through Dynamic Programming

All of the solving methodologies used in this thesis assume that memoization is being used, and the (i, j) values in the following definitions show the location in the look-up table the partial solutions are to be stored.

Longest Common Subsequence Problem

Assuming the characters of the first input string (A) are indexed A_1,A_2,A_i,$\ldots$,A_n, and the characters of the second input string (B) are indexed B_1,B_2,B_i,$\ldots$,B_m .

To identify the length of the longest common subsequence between the two strings a table is created which is of dimension $m \cdot n$, where n is the length of the first input string, and m is the length of the second. Each cell in the table $m \cdot n$ contains an integer value denoting the longest possible sub-sequence based on the sub strings from $A_0,\ldots,A_i$ and $B_0,\ldots,B_j$. Therefore it should be apparent that the cell of the table (m,n) will contain the length of the longest common subsequence for the complete two sequences. This is defined in Eqn. 2.12, where *LCS* denotes the longest common subsequnece scoring grid.

$$LCS_{i,j} = \begin{cases} \emptyset & i = 0 \text{ or } j = 0 \\ LCS_{i-1,j-1} + 1 & A_i = B_j \\ \max \begin{cases} LCS_{i-1,j} \\ LCS_{i,j-1} \end{cases} & o.w. \end{cases} \tag{2.12}$$

A dynamic programming based algorithm will work through populating this table beginning at $(1,1)$ and moving through the table filling a row at a time. If the character of string A at location i matches the character of string B at location j, the running total of *'subsequence recorded so far'*, based on the previous count stored at $i-1$ and $j-1$, is incremented and updated [94]. In this manner the table can be filled using solutions to previous problems, satisfying the dynamic programming requirement of overlapping sub-problems. This approach is a bottom up method of dynamic programming.

Once the table has been filled, as well as having the length of the longest possible subsequence, it is also possible to trace back through the table to rebuild the actual sequence of characters. This is due to the fact it is possible to identify which characters are present in the subsequence, as it is on these characters that the count was updated. Therefore by beginning in cell (m,n) and tracing back to each point the count was incremented the longest common subsequence can be built, or if there are multiple LCS's of the same length, a single one of these will be produced. The definition of this is Eqn. 2.13, where *LCSSTR* denotes the scoring grid with the output from the longest common subsequence algorithm. With adaptation this relationship can be altered to produce many sub-sequences if there is more than one, but this is not discussed here.

$$LCSSTR_{i,j} = \begin{cases} \text{“ ”} & i = 0 \text{ or } j = 0 \\ LCSSTR_{i-1,j-1} \cdot a_i & x_i = y_j \\ \begin{cases} LCSSTR_{i,j-1} & LCS_{i,j-1} > LCS_{i-1,j} \\ LCSSTR_{i-1,j-1} & o.w. \end{cases} & o.w. \end{cases} \tag{2.13}$$

Edit Distance Problem

The edit distance problem can be solved using a similar procedure to the methodology presented for the longest common subsequence problem, as they are closely related.

$$\begin{aligned} d_{i0} &= \sum_{k=1}^{i} w_{del}(b_k) \\ d_{0j} &= \sum_{k=1}^{j} w_{ins}(a_k) \\ d_{ij} &= \begin{cases} d_{i-1,j-1} & a_j = b_i \\ \min \begin{cases} d_{i-1,j} + w_{del}(b_i) \\ d_{i,j-1} + w_{ins}(a_j) \\ d_{i-1,j-1} + w_{sub}(a_j, b_i) \end{cases} & a_j \neq b_i \end{cases} \end{aligned} \tag{2.14}$$

As before strings are stored as sub strings of increasing size, allowing partial solutions to the problem to be stored in the scoring table, and reused in the future to accelerate computation. Again, the scoring table is of size $m \cdot n$ and each cell at (i, j) contains the edit distance thus far for the sub strings $a_0, \ldots, a_i$ and $b_0, \ldots, b_j$. The final edit distance can be found in cell (m, n). As with the longest common subsequence, this is a bottom up method of dynamic programming.

This is shown in Eqn. 2.14, where *d* represents the scoring grid for the edit distance problem, *w* the function which returns the cost of the different available operations on the strings, and *a* and *b* the two input strings.

The key difference between the edit distance problem and the LCS problem is in this case we are searching for when the string changes, rather than when the string remains the same, as is represented in the second case clause. In this second case, when the string is altered, the algorithm should seek to identify the transformation that must take place, in order to transform the first string to the second, and applies the transformation with the least cost. It is at this point that the cost of each operation can be changed such that different scoring systems for the edit distance problem can be implemented.

Knapsack Problem

Both of the knapsack problems already defined can be solved using dynamic programming.

Unbounded Knapsack Problem

$$m_i = \begin{cases} 0 & i = 0 \\ \max\limits_{w_i \leq w} (v_i + m_{w-w_i}) & o.w. \end{cases} \tag{2.15}$$

0-1 Knapsack Problem

$$m_{i,w} \begin{cases} m_{i-1,w} & w_i > w \\ \max \begin{cases} m_{i-1,w} \\ m_{i-1,w-w_i} + v_i \end{cases} & w_i \leq w \end{cases} \tag{2.16}$$

In the case of the formulation of the unbounded knapsack problem presented in Eqn. 2.15 the scoring table which contains the partial solutions, is in fact a vector of length W, where

W is the capacity of the knapsack. The dynamic programming definition states that each cell w of the vector m is storing the maximum profit that can be attained when the capacity is w, therefore the final solution to the problem is present in cell W. Note this definition is only applicable when the weights are strictly positive integers ($w_i > 0$), and the solution is simply an attainable profit value rather than a set of items.

The dynamic programming definition for the 0-1 knapsack problem differs from the definition for the unbounded knapsack problem [90], as the constraint that each item can only be selected once must be maintained. Therefore a two dimension scoring grid is created, with a width of W and n rows, recalling that n is the number of items being considered. This allows for the calculation of the effect adding an item will have on the current total profit, based on the current free capacity. As each item is considered rows are traversed, with the current running total being stored in the scoring grid. Therefore, at the end of computation, the final available profit is stored in cell $(W.n)$.

The Travelling Salesman Problem

Representing this problem using dynamic programming is slightly more challenging than the previous problems. Firstly, assuming city x as a start point: for every other city i we find the minimum cost path with x as the starting point, i as the ending point, and in which all cities appear exactly once. Let the cost of this path be defined as p_i. Therefore, the cost of the cycle would be $p_i + c_{i,x}$. Thus, the optimal tour is $\min_{\forall i \in \{0,1,\ldots,n\}} (p_i + c_{i,1})$, where n is the total number of cities. This means that rather using a single scoring grid, p_i is calculated through the creation of multiple simultaneous scoring grids, S, which are created and filled through a recursive relationship. It is due to this that both the computational and memory complexity of the TSP problem is so high.

Assuming that 1 was the start and end city for the desired tour, S_{Xi} is defined as the minimum cost path visiting each vertex in set X exactly once, starting at 1 and ending at i. This can be represented using the following recursive case:

$$S_{X,i} = \begin{cases} c_{1,i} & |X| = 2 \\ \min\left(S_{X\setminus i,j} + c_{ji}\right) & j \in X \wedge j \neq i \wedge j \neq 1 \end{cases} \tag{2.17}$$

All Pairs, Shortest Path Problem

The all pairs shortest path problem can be solved through a dynamic programming methodology, the Floyd-Warshall algorithm [29]. The principle of this algorithm is based on the fact the shortest path from i to j will be the shortest path from i to k then k to j, where k at some point becomes every vertex.

Therefore, in this algorithm instead of simply iterating over the grid once to fill each cell (i, j), an additional inner loop k is also required to iterate over the scoring grid repeatedly in order to update all pairs of paths until the shortest is identified. This solving implementation is a top down approach to dynamic programming where i, j denote indices within the scoring grid:

$$m_{i,j} \begin{cases} m_{i,k} + m_{k,j} & n_{i,j} > m_{i,k} + m_{k,j} \\ m_{i,j} & o.w. \end{cases} \text{for } 1 \leq k \leq |V| \tag{2.18}$$

Obviously, the size of the scoring grid required for maintaining the paths is the same size of the original adjacency matrix (vv), as the output is an updated adjacency matrix containing the shortest paths. This leads to the time complexity of the algorithm being $O\left(v^3\right)$ and the memory complexity of the algorithm as $O\left(v^2\right)$.

2.3 Alternative Solving Methods

In this section we briefly consider methods to solve the introduced problems, other than dynamic programming, for both completeness of this thesis and to justify why these methods have not been selected as the focus of this work.

2.3.1 Inexact Methods

Inexact methods are ways of solving the problem that provide an solution to the problem, but with no guarantee of finding the optimal solution. Our primary motivation for moving away from such methodologies, and indeed this body of work, is we believe that the extremely large amount of computational power offered by GPUs extends the usefulness of exact methods to an extent that would have seem inconceivable a few decades ago.

Inexact Heuristics

Inexact heuristics are essentially *educated guesses*, which, usually after being applied repeatedly, produce valid solutions to a problem, but have no way of guaranteeing the optimality of these solutions. Due to this, they are popular methods of solving large scale optimisation problems, as they can often provide valid solutions quickly. Often, for large problems, solving exactly is impossible and therefore there is no alternative to using a method such as an inexact heuristic. A meta-heuristic is simply a process that guides the development of the heuristic, i.e. in some way it seeks to improve the guessing process over time. Through the use of an ideal meta-heuristic, it should be possible to iteratively improve a solution over a given time frame, therefore allowing a large problem to be processed for a fixed amount of time producing a good answer at the end. It is also possible for an inexact heuristic to find the optimal solution, but there is no way of validating this unless the problem has a prior know optimal solution. Naturally however, exact methods are preferable to these, as optimality is guaranteed. Examples of meta-heuristics include evolutionary algorithms, tabu search and simulated annealing. There are a range of existing works on both the tabu search [36–38] and simulated annealing [1, 55, 88], including parallel versions for both the CPU [28, 32, 6] and the GPU [85, 27, 46]. Due the level of existing investigation relating to these approaches, finding a research niché in this area would be challenging, thus this thesis seeks to move in a different direction. Detailed information on these methods is not provided here, and the reader is referred to the existing literature.

Evolutionary algorithms are an extremely popular meta-heuristic, very commonly used in optimisation, and as such some preliminary investigation into these was carried out to evaluate their effectiveness, as well as the potential for parallelism. The basic structure of a genetic algorithm is igiven in Alg 1. The premise is to begin with valid solutions generated either at random, or through some form of heuristic, which are continually combined in an effort to improve the solution quality. As improved solutions are found, they replace old solution which are not as promising, mimicking real world evolutionary principles.

Algorithm 1 The basic outline of a genetic algorithm

```
Generate a set of random solutions, the population
repeat
    Select $n$ random solutions from this set, the parents
    Combine these solutions, the crossover to create a new solution, the offspring
    If the new solution is better, replace $m$ parents with it
until stopping condition
```

Suitable methods of implementing these algorithms in parallel were investigated with the most common being either breaking the population into smaller parts and running smaller EAs in independence, and exchanging population members at intervals, or simply evaluating the quality of the updated population in parallel each generation. After implementing these algorithms in parallel on a CPU based system in the early stages of the development of this work, we identified there is already a wide range of GPU literature on implementing both such methods in parallel [79, 100, 46], and the motivation for the work was not as strong, as such the investigation was taken no further.

Polynomial Time Approximation Schemes

A polynomial time approximation scheme (PTAS) [91] is, as the name suggests, an algorithm that produces an approximate answer to an optimisation problem in polynomial time. A PTAS seeks to solve an instance of the problem based on the parameter ε (where $\varepsilon > 0$), and produces a solution to the problem that is within a factor of $1-\varepsilon$ (or $1+\varepsilon$ for minimisation problems) of being optimal. The time complexity for a PTAS can be different for differing values of ε, for example as the amount of precision the algorithm maintains is increased or reduced, but must always be polynomial in terms of n [47].

A more specialised type of PTAS is a fully polynomial time approximation scheme, an FPTAS. In this case there is a greater restriction on the run-time complexity class, where it must be polynomial both in terms of n and $\frac{1}{\varepsilon}$.

Finally there is an alternative type of approximation algorithm, the polynomial time randomised approximation algorithm (PRAS) which seeks to find solutions that are have a high probability of being ε optimal. Obviously the accuracy of the algorithm is dependent on the threshold of the probability of being ε optimal. As with the FPTAS in the previous case, there is also a fully polynomial time randomised approximation scheme, which enforces the same constraint of requiring a complexity that is polynomial both in terms of n and $\frac{1}{\varepsilon}$.

2.3.2 Exact Methods

The most obvious exact method is a naive brute force method, which simply tests valid solutions to the problem until they have all been tested, thereby ensuring the method is exact. However, this method is extremely computationally intensive and therefore infeasible for all but the smallest problem instances.

Exact Heuristics

As well as inexact heretics which quickly generate valid, but not guaranteed to be optimal, solutions to problems, heuristics can also be exact.

For example in the case of a route finding algorithm - if a heuristic was returning the shortest path between pairs of nodes, and in some instances this was pre-computed and known data, the heuristic would be returning exact results. Therefore an algorithm guided by this exact heuristic would have guaranteed optimality.

Also, when the portion of the algorithm a heuristic is responsible for calculating becomes so small, it can return exact results. Using the example of a graph based algorithm - let us assume the heuristic is responsible for calculating some value based on groups of nodes. Should these groups become small enough for the heuristic to perform an exact calculation, it is now classed as an exact heuristic.

Algorithms can have combinations of exact and inexact heuristics, or heuristics which change classification based on the specific input size passed to them. However, only when a heuristic algorithm is guided by solely exact heuristics, can optimality be guaranteed.

Branch and Bound

A common method of solving optimisation problems is the *branch and bound* approach. Due to its prevalence and ability to be adapted to a wide range of problems, as with evolutionary algorithms, at the beginning of this research some time was spent investigating these.

The basic premise of a branch and bound algorithm is that all the possible solution paths are traversed, the branching, which ensures the optimality of the solution and the exactness of the algorithm. However, when a path becomes less promising than others, the traversal stops and it is removed from the search space, the bounding. The first step of running a branch and bound algorithm is to generate a lower bound, a value to which the actual answer to the problem can be no worse than. There are many different way of doing this, either through a heuristic or simply generating a random solution. From this point the a tree is constructed, in which all the solutions to the problem can be built iteratively, however at each node an upper bound is determined which is an estimate as to ‘the best possible solution that will be in this branch of the tree’. Then at each node, before creating children from it, it is evaluated whether the best solution of the branch is worse than the current lower bound, and if it is, the branch is pruned and simply not created. Therefore it is crucial that the bounds are pessimistic to the quality of the solution in the branch, as if they return results that are too high, valid branches may be cut which will break the exactness of the algorithm. As the tree is built, the quality of the best solution found so far is recorded and updates the lower bound during the execution of the algorithm, therefore allowing branches to be pruned more efficiently as the algorithm progresses.

Some preliminary work was carried out investigating these, and again some CPU parallelisation work was carried out. In our investigations we identified that the primary limiting factor of these was the need for constant communication across the processors in order to maintain the solution quality. This is due to the fact when a branch finds a new improved solution and wants to update the lower bound, all the other processors need to be aware so that they can prune there branches appropriately. However, having large amounts of communication in a parallel program can hurt performance, as discussed in the next section, but similarly if less frequent communication takes place the less efficiently the algorithm will run and the more wasted work is taking place, so a balance between the two factors needs to be struck.

2.4 Parallel Programming

> “Highly parallel computing architectures are the only means to achieve the computational rates demanded by advanced scientific problems.” [2]

Parallel processing and parallelism are at the heart of this thesis, thus it is important to understand the basic concepts before presenting our main research contributions in detail.

Generally, programmers not involved in high performance computing used to rely on advances in the clock speed of a processor, and the performance of the hardware in order accelerate programs they developed. However, in the early 2000's advances in improvements of the speed of the processor slowed due to heat and power limitations. Therefore chip developers moved towards a model of developing processors which included *more* cores rather than simply faster cores. This led to new programming challenges and the emergence of parallel programming [87].

2.4.1 Parallelism

Parallel programming is the term used to describe when a program is designed to run on more than one processor in a computer simultaneously. This is achieved by breaking the program down into smaller components that can run in independence of each other, communicating through fixed channels. Implementing a program using parallel programming is desirable, as it has the advantage of improving (reducing) the run-time. This means the program will simply take less time to run, or larger problems can be dealt with in the same time frame as before. For the purposes of this thesis, we are interested in parallel programming as it may allow the solving of larger problems in an exact manner, something that would not be feasible without it.

The way in which the program is broken down into the individual components is heavily dependent on the algorithm being implemented, the data which it is to process, and the architecture or environment in which it is to be deployed. However, it can be classified into two broad categories; *task parallelism*, in which separate parts of the algorithm run concurrently and *data parallelism*, where the same part of the algorithm is operating on multiple pieces of the input data simultaneously. Generally however, programs use elements of both paradigms, and fall somewhere on a spectrum between the two. At this point we also define the term *execution pipeline*s. Pipelines enable a level of parallelism, by allowing multiple pieces of data into the pipeline simultaneously. For example, a piece of data could have an operation performed on it by stage one of the pipeline, while a different piece of data has a different operation performed on it by stage two, in a single clock cycle, thus allowing operations to be carried out in parallel.

As NVIDIA CUDA, which this thesis uses, adopts primarily a data parallelism paradigm, this is discussed next.

Data Parallelism

Data parallelism is the process of breaking down large data into smaller blocks, and passing these blocks to independent processors to process simultaneously. . Therefore this is only applicable in situations when different sections of the input data have no relationship to each other. For example, consider our case of a dynamic programming scoring grid. Should a processor need data from a different cell of the scoring grid in order to process the current cell, but this second cell is not in the same block the processor has been allocated, an error can occur if this is not handled. To handle such a scenario, communication will be required between the two processors, which can often be a slow process. This means the viability of parallel data processing is heavily dependent on the structure of the algorithm, as well as the relationship between the different points of input data. The size of the blocks, and the granularity of how far the data is broken down can be altered by the programmer as they see fit, for the algorithm and the architecture the program is to be deployed on. Should sufficient processors be available, or a massively parallel architecture be used, it is possible to break the data down such that each processor is allocated a single piece of data. In our example of a scoring grid, each processor is allocated a single cell of the grid.

In nearly all parallel processing settings it is required that there should be a level of communication between the independent processors. Examples of this could be updating each processor with information about progress in other blocks or synchronising some shared value between processors. A common cycle of a parallel program is to do a fixed amount of work, pause for communication, then continue working. There are two different approaches to dealing with communication in a parallel setting; explicit communication through a message passing system or implicit communication through the blocks sharing a portion of shared memory which multiple processors have access to.

Implicit communication through shared memory is the communication method which NVIDIA CUDA adopts, and therefore the method which will be adopted during development of the model in this thesis. Communication via shared memory takes place through multiple threads monitoring the same area of memory and watching for changes. A simple example would be, if communication was to occur, the processors would monitor the state of some shared variable, and when it transitioned to a specified state, they would be aware that other processors had written data for communication to the shared area. At this point, other processors would be aware it is safe to read this data back.

In a parallel setting, the time taken during communication can be a bottleneck in the code, as at these points the processors all have to cease computation, and wait until the communication finishes. This is an issue that is magnified when processors may be at different stages of work, and as such one may be idling for a considerable amount of time whilst it waits for another to finish and be ready to communicate. Therefore, for a parallel program to be effective, the amount of communication should be kept to a minimum.

As dynamic programming focuses on the idea of a central scoring grid, naturally dynamic programming algorithms would tend towards data parallel approaches.

2.4.2 NVIDIA CUDA

> "In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU." [71]

> "The computing industry is at the precipice of a parallel computing revolution, and NVIDIA's CUDA C has thus far been one of the most successful languages ever designed for parallel computing." [83]

Graphics processing units have been developed to allow the high speed, real time rendering of 3D objects, primarily for the use in computer gaming, but also in visual computing and design applications. Due to the need in computer games to perform a massive number of floating point operations per second, during the rendering of frames, this has pushed GPUs down the route of massively parallel processing where they are equipped with extremely large numbers of small processing units that can perform data parallelism very effectively. Nearly all modern GPUs have hundreds of processors, with higher end variants pushing that figure to several thousands.

In 2006, NVIDIA released the first version of the Compute Unified Device Architecture which allows the programming of GPUs for uses other than graphics, which led to the rise of general purpose GPU programming (GPGPU). Originally CUDA was only available through official C bindings, but since then C++ bindings have been added as well as third party support for a variety of languages such as Java and Python. In 2010 OpenCL [86] was launched, a language that seeks to offer the same functionality without being proprietary to one hardware manufacturer. However, we choose not to use OpenCL in this thesis as it adds additional complexity during programming when dealing with the generic wrappers, and when running OpenCL code on a NVIDIA card, the compiler simply auto translates it to CUDA first.

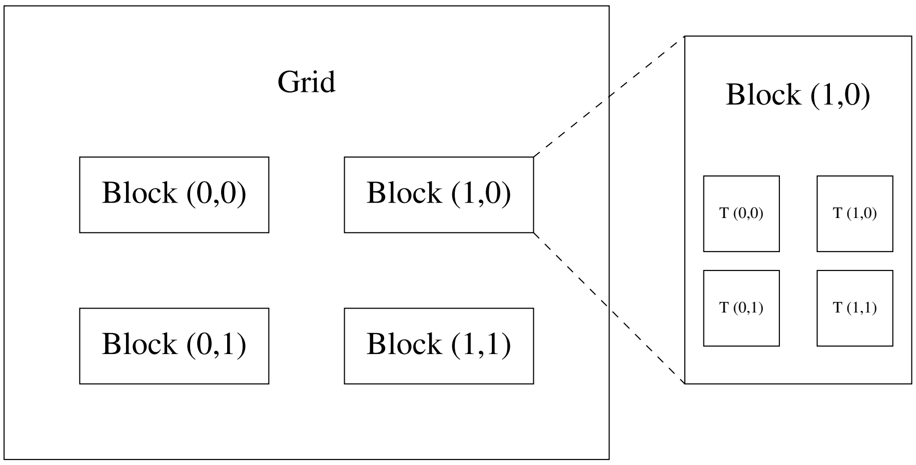


Fig. 2.7 A representation of the block and thread hierarchy adopted by the NVIDIA CUDA. T denotes individual threads

As aforementioned, CUDA adopts a heavily data parallel approach, and uses a pipeline execution architecture. This means that adding divergence to the code, i.e. `if` clauses and branching, can introduce a lot of inefficiency to the program as both sides of the branch must be evaluated in the pipeline. When programming an application to run on a GPU through CUDA, the programmer interacts with the available threads of the GPU through a *grid*. This contains a series of *blocks*, and each block contains a group of threads. This is shown in Fig. 2.7. The indices applied to each block and thread are used for the programmer to reference them, however they have no direct mapping to the underlying hardware. Instead they are used as a programming aid, as it is often possible to define some form of mapping between the input data and the indices of the underlying block, allowing for clear implementations of how the parallelism is applied to the data.

The blocks are executed on independent multi-processors of the GPU, with new blocks being loaded as multi-processors become available. There is no guarantee as to the order in which blocks are loaded, so it is critical when writing CUDA programs there is no dependency between separate blocks. Once execution has begun on a block, it cannot cease until the block is complete.

As the clock speed of the CUDA architecture is so high, the memory on the GPU can often become the bottleneck of the execution due to its relatively high latency. On a GPU there are several different types of memory, all of which have been optimised for a specific task. The largest memory store, and also the slowest by a considerable amount is global memory, which is available to all blocks and threads at any given time. Moving down the hierarchy there is a smaller amount of faster shared memory, which is visible between all threads within a block. Finally, the fastest memory on the GPU is the highly limited thread local memory, and the even quicker and highly limited thread local registers. Additionally, the GPU also has specialised memory stores the programmer can hinge; constant memory and texture memory, both of which are global stores. Constant memory is optimised for broadcasting the same data to multiple threads simultaneously, and texture memory is optimised for requests for data that are spatially close.

Compared to the high performance of the GPU cores, GPU memory is relatively slow. Therefore steps have been taken in the CUDA architecture of CUDA to compensate for this. Once execution has begun on a block it cannot be then unloaded, however during execution blocks are broken down into smaller units of threads called *warps*, which are processed by the multiprocessor. Should a warp be requesting a memory operation, which is high latency, the scheduler can context switch this warp out and bring another one into the multiprocessor to be processed. In this way the latency of the memory can be hidden as long as there is sufficient parallelism to keep the multiprocessors saturated. The concept of warps is also linked the issue of divergence on the processor. As the smallest unit of threads that can be dispatched to the multiprocessor is a half warp, and in pipeline architectures all threads are required to follow the same code path, if only a small number of threads in the half warp require this code path, the rest will simply do nothing affecting efficiency. Therefore having divergence, and nested divergence in a CUDA program can have a huge impact on the efficiency of the execution.

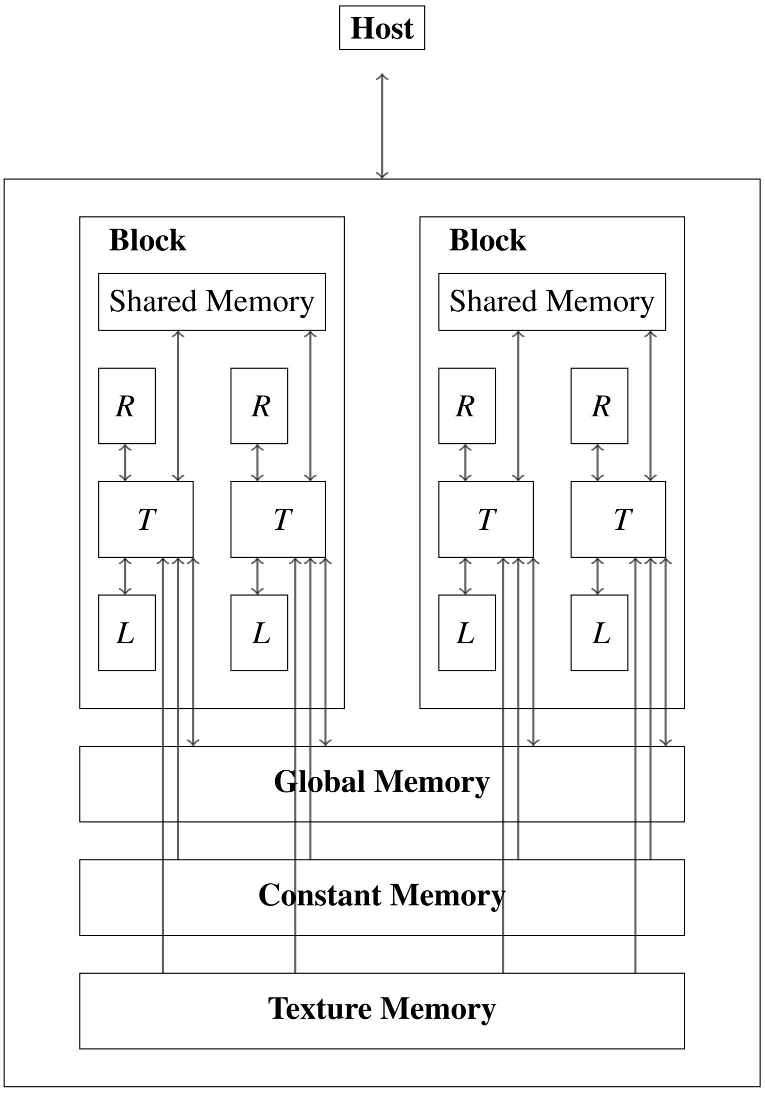


Fig. 2.8 The memory hierarchy of a graphics processing unit, and how these different memory stores interact. R = registers, T = thread, L = local memory

All work is dispatched to the GPU within a *kernel*, which is analogous to a function which to be executed on a GPU. Once the kernel has been dispatched, the CPU has no direct interaction with it, and can only monitor it's progress by requesting the state of the GPU from the CUDA API calls. More modern implementations of CUDA, coupled with modern hardware allow the running of more than one kernel simultaneously on a GPU, or allow the dispatch of new kernels from ones currently executing enabling some level of recursion. In all cases from a programmers perspective the launch of a kernel on the GPU is a non-blocking operation, freeing the CPU to perform other tasks whilst the GPU is processing. This is has led to the rise of *'heterogeneous computing'* [65, 15] where both the CPU and GPU(s) of the system are running in tandem to ensure that maximum efficiency is drawn from the computing resources available. Obviously however, care must be taken when dealing with multiple kernels, or using the CPU at the same time, as due to the fact the kernel launch is not blocking it is easy for synchronisation issues to appear.

As CUDA is a data parallel architecture there are limited options for synchronisation, as the paradigm is geared toward performing the same operation across large data sets in no guaranteed order. CUDA does provide the ability to synchronise threads within the same thread block, but not to synchronise across the entire GPU. This is by design to ensure that programmers are creating blocks that are not dependent on one another, and the scheduler is free to load and drop blocks as it wishes. Dispatching a new kernel is a very quick operation, and if global synchronisation points are required, the programmer should use the launch of a new kernel to serve this purpose, and ensure the CUDA paradigm is not violated.

Due to the underlying inherent architecture of the GPU there are some use cases in which there is a potential for a large amount of speed up, and similarly there are cases where it may not be beneficial. As GPUs were originally designed for graphical operations, it is no surprise that video or image manipulation are some of the most promising candidates for speed up on the GPU, able to achieve speed-up factors hundreds of times over serial code. Where as an algorithm with a complicated execution flow, and a large amount of dependencies between steps is likely to not achieve a large degree of speedup, if any. In essence, the more data parallel an algorithm is, the more it is suited to the NVIDIA CUDA architecture.

With dynamic programming using large scale matrices, and potentially benefiting from a data parallelism approach, NVIDIA CUDA is obviously a natural fit for accelerating such algorithms. Next, we consider existing approaches to this already present in the literature.

2.4.3 Existing Parallel Methods

The Longest Common Subsequence

Due to the amount of uses for the longest common subsequence problem, there is a wide amount of existing literature available.

In 1990, Apostolico et al. [5] presented a simple divide and conquer based approach to solving this problem. The parallel model was very coarse grained, meaning that large chunks of the input data were divided between the available processors. The algorithm ran by dividing the entire scoring grid into smaller chunks and dividing these between a small number of CPUs, and these are then computed in parallel (the division phase). However, it still requires a large amount of serial operations to handle the boundary data on the edge of each block, and communication between the individual chunks (the conquer phase). Due to the amount of serial communication, the applicability of this approach is limited to only a small number of processors. The algorithm has a computational complexity of $O(log(m) \cdot log(n))$, where n is the length of the first input string, and m is the length of the second. Whilst this is now a dated piece of literature, it serves to demonstrate that the most obvious method of parallelising the LCS problem is to use a simple divide and conquer approach, and then seek to mitigate the communication and synchronisation issues that arise from this.

Moving on from this in 1994 Lu & Lin [59] propose a method to improve the runtime complexity. They show that an LCS problem can be also be represented as a directed acyclic graph (DAG). The objective of the problem is then to find the maximum weight path from the top left of the scoring grid, to the bottom right, as this will be synonymous with the longest common subsequence. To find this, the algorithm finds the maximum cost path from every vertex on the top row to the bottom. Similar to the previous approach, a divide and conquer approach is used for parallelism, but in this instance it is much more fine grained approach, where the grid is recursively split into an upper and lower portion and solved. This means that the grid can be decomposed all the way to two row chunks, allowing far more processors to be used simultaneously. The complexity of the algorithm is improved to $O(log(m) + log(n))$. This work shows that splitting the grid into smaller pieces unsurprisingly allows for a greater degree of parallelism. It also shows how satisfying large chunks of dependencies at once can improve the parallel efficiency.

This was improved upon again in 1997 by Nandan & Saxena [68] who maintained a very similar paradigm, but through a level of pre-computation, generate what they termed as a *cost matrix* for each pair of rows. Whilst this is a serial portion of the code, it means that non promising paths can be removed much more quickly, and therefore considerably reduce the overall amount of computation required. In this case the complexity of the algorithm is now reduced to $O(log(m))$. This paper demonstrates that by allowing a small amount of computation to take place before the main execution begins, run-time can actually be improved, so optimisations should not just be sought in the main algorithm.

In 2005 a parallel algorithm was proposed by Xu et al. [97] for implementation on an optical bus. Whilst this is a highly specialised architecture, it has the benefit of being quite similar to the architecture that the GPU implements i.e. many small processors which form a pipeline. Also optical bus systems heavily rely on multi-casting for communication which is analogous to a GPU broadcasting a message to each processor in a block. In this work a very fine grained form of data parallelism is used, where each cell of a row is calculated in parallel before the same then occours in the next row. Whilst this is a simplistic method of parallelism without the need for any pre-computation nor an advanced synchronisation method, it takes advantage of the very high number of processors available; a method such as this would be suitable for implementation upon a GPU. The computational complexity of this parallel algorithm is $O\left(\frac{m \cdot n}{p}\right)$. This work was very interesting in the respect that it shows when using a very high number of processors, and a SIMD architecture, it is possible to run a dynamic programming algorithm in parallel using the simple approach of allocating each cell to a processor and synchronising when appropriate.

The last CPU work we consider is from 2006 by Krusche and Tiskin [54]. This paper details a highly efficient method of moving through the scoring grid as a *wavefront*, where cells are calculated in parallel as the dependencies are satisfied. This means that cells are filled in a diagonal method from the top left of the grid, down to the bottom right, as dependencies allow successive cells to be filled. Also presented are some algorithmic optimisations such as compacting multiple cells of the grid into successive processor words to decrease memory bandwidth requirements and speed up execution. This lowers the computational complexity again to $O(n)$. The work detailed here shows when attempting to parallelise a dynamic progrmming algorithm it can be very effective to traverse across the grid in a wave like method, simply filling cells as soon as possible as the dependencies for them become satisfied.

There has also been work on implementations for the GPU. Kloetzli et al. [53] demonstrate a diagonal wave front based implementation similar to Krusche and Tiskin [54] running on a GPU. This is an appropriate model to deploy as it is a highly data parallel approach, with each cell being calculated in parallel by an individual GPU core, and due to the diagonal nature of the wave front, large numbers of cells can be computed at once. However, it should be noted that this method of calculation leads to periods of *warm up* and *warm down*, which are times when the device is not fully utilised at the start or end of execution because the size of the wave front is smaller than the available number of cores. Also detailed is how this would scale into a multi-GPU setting, using native CUDA functions, where the blocks are mapped across many GPUs as opposed to a single one. We feel this is a very important work as it demonstrates that a diagonal wave front is highly effective on the GPU, using the available computational resources , whilst also mapping to the dynamic programming paradigm effectively. Also, this may be appropriate to more problems, as it does not require any problem specific features beyond the identification of the dependencies of each cell. The discussion about multiple GPU implementations was interesting, as it demonstrated how powerful the CUDA API is needing little additional work to scale this beyond a single GPU.

Extensions have followed on from this work, such as adding an element of pre-computation to allow for a greater degree of parallelism [99], or coupling pre-computation with algorithmic optimisations to maximise performance on the GPU [75]. All of these however follow the same principle of having a wave front traverse the scoring grid in a diagonal manner, such that the amount of parallelism at any given time is maximised, and this is the model that most effectively takes advantage of the massively parallel architecture on the GPU. Whist these algorithms do offer novelty over the previous approach by Kloetzli et al. [53], and it is important to consider and evaluate the points they make, we feel that they do not further the discussion by as much.

As the edit distance problem is so closely related to the LCS problem, and the dependencies are the same, it can be assumed that if an algorithm is valid for the LCS problem it will also be valid for the edit distance problem. Whilst this may not be the case if many specific optimisations have been applied, generally this assumption holds. The same applies to the Manhattan tourist problem.

The Knapsack Problem

In 1988 Lee et al. [56] propose a divide and conquer method that allows the dynamic programming implementation of the knapsack to be implemented in parallel. The overall scoring grid is broken up into smaller chunks, and these chunks are then filled in parallel. As with the early parallel implementations of the LCS problem, this is a coarse grained approach, designed for deployment on a small amount of processors and is only parallelised at a basic level. This still requires serial communication between the blocks, which limits how well it can scale to larger numbers of processors. In this work the authors have opted to present the communication between the blocks in a hypercube structure, in an attempt to make the communication as efficient as possible. This algorithm has a computational complexity of $O\left(\frac{n \cdot c}{m+c} \cdot log\left(m+c^2\right)\right)$ where n is the number of objects, m the number of processors, and c is the capacity. As with the first paper considered for the LCS problem, this initial piece of literature demonstrates that a simple divide and conquer based approach to breaking down the dynamic programming grid is valid, which then only needs a communication method between these to be designed.

Lin & Storer built upon this in 1991 [57], by restructuring the overlapping sub problems within the dynamic programming definition. In this work, the knapsack problem is formulated using the multistage representation [16]. This means that whilst the amount of dependencies required are the same, the computation of each element is independent of one another allowing for more efficient parallelism. The computational complexity of this algorithm is improved at $O\left(m \cdot c \cdot log\left(\frac{n}{m}\right)\right)$. For a small processor count it is possible that the approach by Lee et al. [56] is faster, than a high processor count using this algorithm, but when the number of processors is the same this algorithm should always be faster. Also, this algorithm allows for the possibility of a greater number of processors. This paper presents the interesting point, that restructuring the sub-problems can allow for a greater degree of parallelism.

Also in 1991 Gerasch [33] presents a parallel polynomial time approximation scheme which has a computational complexity of $O\left(n \cdot log\left(\frac{n}{\varepsilon}\right) + n + log\left(n\right)\right)$. This is based on the two list approach to solving the knapsack problem [44] which moves away from the dynamic programming approach we are seeking to use. In this algorithm, the first step is to sort the items by profit and weight before computation begins, and this can be parallelised. The authors use a simple parallel prefix sum operation to achieve this. Also, the core dynamic programming loop is implemented in parallel; as this is an approximation scheme, multiple feasible solutions are calculated, which in this algorithm are produced simultaneously. We have not considered a PTAS before this point, although the paper clearly demonstrates that the generation of multiple solutions within a PTAS naturally lends itself to being implemented in parallel. However, as a PTAS does not generate an optimal solution, its relevance to the work in this thesis is limited.

In 2004, Goldman & Trystam [39] demonstrate an advanced divide and conquer algorithm. They show how to restructure the scoring grid as a precedence graph, which is a specialised type of directed acyclic graph, and is used to represent the dependencies within in the algorithm. To solve the problem, paths through the graph are then calculated in parallel, and an adapted backtracking algorithm can then be used to find the final solution. The nodes of the graph maps to an irregular mesh and the communication across this is maintained through a hypercube structure. We feel that this work is quite similar in its design to the LCS paper by Lu & Lin [59], with the concept of taking the scoring grid and remapping it as a graph. We found the work interesting, as it goes into great depth to consider the dependencies between the cells of the grid, as well as discussing different way in which this can be represented.

While reviewing the literature we found there are limited implementations that consider the classic dynamic programming case, with many choosing instead to focus on branch and bound [18, 25, 58] implementations, as well as inexact heuristic methods [69, 20].

In 2011, Boyer et al. [13] proposed a similar algorithm, but refer to their implementation as a *dense* dynamic programming method, as they have gone to great lengths to compact data, in an effort to minimise and reduce memory transactions as far as possible. Also in this implementation it is demonstrated that this algorithm can be deployed on multiple GPUs simultaneously, allowing for even greater degrees of parallelism. The authors achieve this by having a single CPU thread control the multiple kernels, and also run a load balancing algorithm on this thread such that a GPU is not left idling for an extended period of time. This piece of work provided a lot of insight into the very fine details of implementing these kinds of algorithm effectively on the GPU, as well providing detail on multi-GPU implementations. We found it interesting that such complex load balancing was required to implement this effectively, and it was not as simple as dividing the work evenly across the GPUs.

Moving onto GPU parallel implementations, in 2012 Boyer et al. [14] propose a parallel method which executes the core dynamic programming loop in parallel, filling a row of the scoring grid at a time, as the dependencies allow. Each cell of this row is assigned to an individual thread, and GPU core, which leads to a very fine grained parallel approach. Also in this work, they consider some of the challenges associated with GPU programming. As dynamic programming requires a large amount of memory, algorithmic optimisations have been applied such as compressing the multiple cells of the matrix into single 32 bit values, and copying data to the host when possible. The authors have identified that host copies are inherently expensive, and have tried to minimise and coalesce them, at natural synchronisation points in the algorithm. Also, the authors have tried to limit the amount of the scoring grid that is present in the GPU at a given time, holding pieces that aren't currently required in the host memory. Finally, how to manage the large number of threads and deal with appropriate synchronisation between is also detailed. We feel this paper has a lot of similarities with the wave front approach put forward in some of the LCS literature [54, 53], where there is a very high degree of parallelism and the scoring grid is simply filled as fast as possible whilst maintaining synchronisation. The points about compressing the memory, and carefully considering each memory transaction were also interesting, and something that had not been considered in such depth in the LCS literature.

All Pairs, Shortest Paths

Beginning in 1987, Jenq & Sartaj [48] present simple a divide and conquer approach. This decomposes the dynamic programming scoring grid into smaller chunks, and manages the communication between these using a hypercube style structure. Whilst this is a fairly naive approach the authors have given some consideration as to how to effectively break down the grid, and how to map these chunks onto multiple processors. We note the literature survey for this problem begins the same way as the previous two, with a decomposition of the scoring grid and using a hypercube for communication. Whilst this literature is now dated, it does demonstrate that a single solving methodology should be applicable for all the problems presented thus far.

In 2003 Venkataraman et al. [92] propose a cache efficient implementation. Whilst this is not actually a parallel approach, it is used to effectively support future parallel implementations so is considered here. In this algorithm a divide and conquer approach of breaking down the scoring grid is used, albeit a somewhat more advanced method than simply breaking the scoring grid down into smaller chunks. Once the grid has been decomposed, the sub-grids are processed in a specific order as part of a three-phase process. Initially sub grids that have no dependencies begin processing, followed by sub grids which only have dependencies on one other which are now satisfied, finally updating the remaining sub grids that have dependencies on two others. We feel this paper is so interesting as not only does it show decomposing the larger grid into sub grids, it discusses how to configure the size of the sub-grids to be optimal for the amount of cache available; a concept we believe will be transferable to individual GPU blocks.

Following on from this Bondhugula et al. [12] in 2006 demonstrate how to implement Venkataraman et al. [92] on a field programmable gate array (FPGA). Beyond some small adjustments to take into account the implementation on a FPGA, no major changes are proposed over the original algorithm. We found this work very interesting, as it overcomes one of the challenges of implementing the APSP problem in parallel, which is the dependencies for each cell span the entire scoring grid. Whilst the multiple stage system of the algorithm may lead to extra work over a classical serial implementation, computation based operations are so cheap and memory operations are so expensive, it is a clearly a beneficial trade off to make.

In terms of GPU implementations, the seminal work on the APSP was in 2007 by Harish & Narayanan [42]. This paper seeks to outline a general algorithm for implementing graph problems on the GPU, of which the APSP problem is a suitable candidate. As this work seeks to deal with graph problems, rather than dynamic programming, it moves away somewhat from the concept of scoring grids and many dynamic programming principles. In this work, a breadth first based traversal method is adapted for use on the GPU, where each vertex of the graph is assigned a thread, and as the graph is traversed synchronisation occurs at each level of the tree. The authors present pseudo-code for solving the APSP problem with this algorithm, however they note there is a restriction placed upon the amount of vertices that can be calculated on the GPU, and indicate this to be around a few thousand. We feel the work in this paper moves away from the classic dynamic programming principles we are focusing on to be of relevance to the work in this thesis, but as it appears to be the first GPU implementation it should be reviewed here.

In 2008, a more specialised APSP algorithm was proposed by Katz & Kider [51] based on the work by Venkataraman et al. [92]. This algorithm is closer to the dynamic programming representation, and as it is targeting the specific APSP performs more efficiently. Also presented in this paper is the memory configuration used which demonstrates how to only maintain the appropriate sub-grids in GPU memory at a given time, based on the dependencies of the ones that are currently being processed. Some discussion is provided on how this would map to multiple GPUs; due to the fact the blocks are so independent between phases, and there is so little communication it should be easy to scale this algorithm beyond a single GPU. We feel this paper is the current reference implementation for implementing the APSP on a GPU. Furthermore, the discussion on linearising of the memory made some useful points, as well as offering insight onto how to most effectively use the different available stores on the GPU.

Most recently in 2014 Djidjev et al. [23] presents a method of using multiple GPUs effectively to solve an APSP instance. This paper is essentially an enhanced divide and conquer method, that seeks to split the input graph more intelligently between the GPUs by looking to exploit natural breaks in the graph, or identify natural clusters of nodes which would be present in real world road networks. By partitioning the graph in such a way, communication between the isolated parts of the graph can be greatly reduced, which in a multi GPU setting can greatly increase the efficiency as communication is an expensive operation. Whilst this paper makes efforts to reduce the communication overhead, we note that the approach presented is not an algorithmic improvement that can be abstracted and applied to other problems, but rather a very problem specific solution.

Dynamic Programming

As this thesis seeks to find a generalised approach to implementing dynamic programming algorithms to solve multiple problems effectively on the GPU, we must now consider more generalised and abstract parallel models.

In 1990 Viswanathan et al. [93] presented a parallel model applicable to various problems, such as finding the optimal order of matrix multiplication and finding an optimal binary search tree. In this paper the authors consider a simple dynamic programming recurrence relation, and describe how the search space can be mapped to a binary search tree. From this point it is possible to break down the search into a series of sub trees which can then be computed in parallel. Through standard parallel reduction techniques the final complete tree can be reconstructed yielding the solution to the problem.

Edmonds et al. [26] discuss a parallel model that assumes a shared memory store is available. In this they propose using the wavefront method (similar to Krusche et al. [54] and Kloetzli et al. [53]), that traverses the scoring grid in a diagonal method, filling cells as soon as the dependencies allow. This work presents the argument from a problem agnostic approach, rather than just being applicable to a specific test problem. However the authors talk about considerable overhead and difficulty from the synchronisation, as this is designed to be deployed on a multiple CPU system. Therefore it is not as simple as assigning each cell to a thread, rather a group of cells across the diagonal axis. At a similar time Galil and Park [30] proposed a very similar method of dynamic programing with differences in the way the grid is broken down and blocked into sub grids along the diagonal axis. We found both these papers to be interesting as they demonstrate that a model we have already found to be effective at parallelism on the GPU, from a theoretical level can be applied to many dynamic programming algorithms.

Tan et al. [89] in 2007, presented a dynamic programming model designed to run on a multi-core architecture, using a string manipulation problem as the test function throughout. The hardware model in this paper was tested on was an *IBM Cyclops64*. Whilst this has not been introduced, the authors discuss the fact that it has large scale parallelism, expensive memory transactions, thread blocks and it is a heavily data parallel model. We believe therefore the architecture this model was deployed on shares many similarities with a GPU based system. In this paper the method still follows the diagonal parallelism approach, however instead of assigning each cell of the matrix to a thread, blocks of cells are assigned to a thread, in a similar method to the CPU approach of Edmonds et al. [26]. The authors then move on to discuss how to effectively use the different speed memory stores, demonstrating more advanced methods for allocating the blocks of cells to threads, as well as giving some consideration to load balancing; something that has not been shown in any of the works presented thus far.

In 2011 Wu et al. [95] extend the diagonal processing method on the GPU to an adaptive method where the level of parallelism changes during execution. In this paper, as there are many cells being processed at once, for example when the wave front is approaching the centre of the grid, then multiple cells are processed by a single thread. However this changes when the wave front is smaller, returning to the paradigm of each thread processing a single cell. In practise the algorithm is far more complex than this, with varying degrees of parallelism, and different patterns of mapping the data to the hardware as the wave front passes through the grid. Using this method of adaptive parallelism, the authors claim that memory transactions can be coalesced more efficiently, as well ensuring that the hardware resources are used as effectively as possible. These optimisations lead to run-time improvements in the order of 10x based on the test problem of optimal matrix parenthesisation. We found this paper interesting as it demonstrates that deploying this class of algorithm on the GPU is not as simple as seeking to maximise the parallelism, but potential factors such as queuing less threads and saturating existing ones more effectively have to be considered. This is also the first example of adjusting the level of parallelism employed as the execution of the algorithm progresses.

Moving onto 2013 this approach was also used by Berger & Galea [10], who as well as using thread grouping to increase the hardware usage per thread, also adjust the level of parallelism based on the underlying GPU hardware available. This is controlled by a series of parameters that the user can define, or is detected automatically by the algorithm at runtime. Whilst the authors use the multiple knapsack problem as a test case, the concepts they present are transferable to other dynamic programming problems. This paper was interesting as the authors specifically stated they are seeking to find algorithmic improvements applicable to the whole spectrum of DP problems, and present improvements based on this premise. The concept of adjusting the parallelism based on the hardware is interesting, but is a common paradigm in parallel programming and is more of a refinement to the algorithm rather than a core feature.

Key Findings

Based on the literature review, we identify some key findings from the literature, and identify shortcomings which will influence the work moving forwards.

- Data parallelism, and divide and conquer, is a highly effective method of implementing dynamic programming algorithms in parallel.
- To calculate groups of cells in parallel, first their dependencies must be satisfied to prevent waiting for communication.
- Traversing through the scoring grid in a diagonal wave front is an effective method of parallelisation for multiple dynamic programming problems, and has been demonstrated on both the CPU and the GPU.
- Steps must often be taken to limit the memory usage on the GPU. A common method of this is to store data that is not required on the host.
- Pre-computation is generally favourable when it will avoid synchronisation or communication later during the execution of the algorithm.
- The literature pertaining to generic approaches to parallelising dynamic programming algorithms pre-date GPUs, and, as far as we are aware, we are the first to tackle this for GPUs.

- Generally all current GPU algorithms target a single problem, or a problem and it's variants, rather than a range of problems.
- All presented papers demonstrate a different approach to managing the memory usage of a GPU implementation, there is no single unified approach.

Summary

This chapter has introduced relevant background concepts to the work that is detailed within the thesis. It has introduced a set of optimisation problems that will serve as test problems throughout this thesis as well as the solving methodologists associated with them. A method of solving problems exactly without using a brute force approach, dynamic programming, was introduced and detailed as forms the foundation of the models which we present later. Also it sought to justify why we have taken this approach to solving the problems, compared to other classic solving methodologies. It has provided a brief overview to the concepts surrounding parallel programming, as well as discussing GPU computing and NVIDIA CUDA the hardware architecture the work presented in this thesis will be deployed on. Finally, it has investigated related work to this thesis to identify current solving methodologists, as well as the short comings associated with these, to demonstrate there is a gap for this research.

Chapter 3

Model Design

Overview

This chapter describes the creation of the parallel model that this thesis is proposing, and the memory structure designed to support it. Also introduced is the method that allows users to define files which describe the high level structure of a problem, allowing the new problems to be defined easily. Small scale benchmarks are also presented as the model is described, which seek to justify the design choices that have been made. Finally, we consider the differences and contribution of this approach when compared to existing methods in the literature, some of which that have been previously discussed in Section 2.4.3.

3.1 Design

Based on the identified shortcomings in the previous literature survey, we now present our proposed parallel model. This model should be high performance, improving the run-time of the algorithm by using the available resources efficiently, and generic enough to allow a variety of optimisation problems to be implemented on the GPU using the same paradigm.

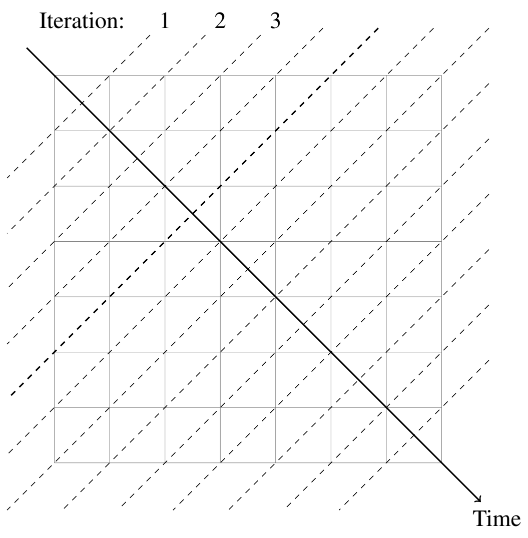
3.1.1 An Anti-Diagonal Approach

A common identified method of parallelism when implementing dynamic programming problems is to iterate through the scoring grid in an *anti-diagonal*, or *wave front* approach [26, 53, 54, 95], and filling the cells of the scoring grid at each iteration in parallel. This is based on the concept that as every cell in a diagonal row of the grid is filled, the dependencies for the next are satisfied, allowing diagonal rows of the dynamic programming grid to be filled in parallel, as discussed in Sec. 2.4.

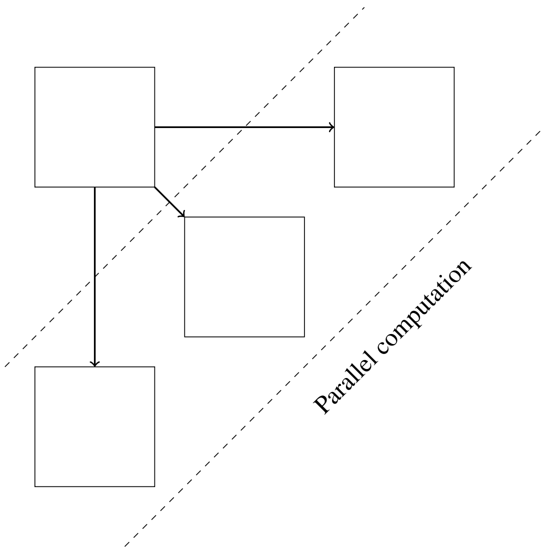
A diagram showing how this traversal method operates is provided in Fig. 3.1a demonstrating the the iterations required to fill the entire grid, and the order in which they occur. The diagram provided in Fig. 3.1b shows that every cell of each iteration can be filled simultaneously, as they are not dependent on each other.

As can be identified from the diagrams, there are going to be periods of *warm up* and *warm down* at the beginning and end of execution, whilst the size of the iteration computed in parallel are increasing and decreasing, meaning all the available parallel resources may not be used. Therefore, achieving 100% efficiency, where all available cores are used throughout the entire execution, will be impossible. However, assuming suitably large problem instances, these inefficient execution periods should only account for a very small proportion of the overall run-time of the algorithm.

We believe that this will be a suitable starting point for our parallel model, because it is a data parallelism approach to parallel computing, which matches the paradigm employed



(a) Iterating through the dynamic programming scoring grid in an anti diagonal fashion. Each dotted line represents an iteration that is processed in parallel.



(b) As a cell being filled satisfies the dependencies of future cells, it allows the elements of a diagonal iteration of the scoring grid, to be calculated and filled in parallel.

Fig. 3.1 How the method of traversing through the dynamic programming scoring grid in a diagonal manner, to allow parallel solving, operates

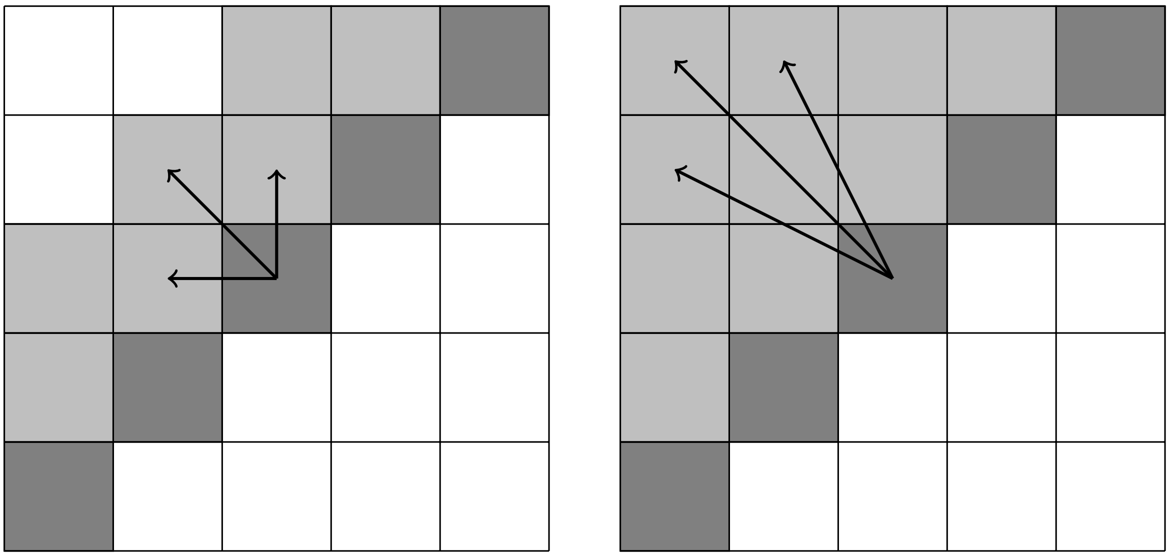
by NVIDIA CUDA. Also as dynamic programming relies on a scoring grid in memory, the model should map to the underlying grid memory structures of the GPU in an efficient manner, and the grid structure should easily divide into smaller blocks to allow for optimal mapping of the data. The size, and dimension of these blocks is likely to have an impact on the performance of the algorithm, and these parameters will be considered geneally in Sec. 3.2, and more specifically for individual problems in chapter 4.

This general approach is unlikely to provide the highest performance for each individual problem, as we have already identified there are specialised parallel methods designed to solve specific problems. However, this is a method that is valid for a whole range of problems, and a suitable starting point for our proposed model, moving us towards the aim of defining a single generic parallel model. We therefore consider this implementation to be a suitable trade off between designing an extremely specialised high performance model targeting a single problem, and a generic model that can solve multiple problems. It is acceptable to assume there will be some performance penalty, and a balance to be struck, when trying to design a model which is suitable for more than problem instance.

It has already been observed from the dynamic programming definitions for the test problems in Sec. 2.2.2, that each problem has a different structure regarding the previous cells of the grid each cell currently being processed is dependent on. For example, the diagram shown in Fig. 3.1 is using the longest common subsequence as the test problem, which is the most appropriate problem for using this methodology. However considering an alternate problem, such as the knapsack problem, to fill cells in this scoring grid it is required that different cells be available rather than simply the previous diagonal iteration. This becomes even more challenging when considering problems such as the travelling salesman problem, or the all pairs shortest path problem, where there are more complex dependencies for the current iteration requiring data more widely distributed throughout the scoring grid.

Details of the specific dependencies across the scoring grid for different problems are considered in more detail in chapter 4, at this point of the design we discuss this issue more generally. Considering the current cell which we are trying to fill, any dependencies that it has in the scoring grid must be satisfied by the time the wave front reaches it. This means whichever memory store is being used to store the main dynamic programming scoring grid, or however this grid will be decomposed to fit on the GPU, dependent cells must always remain present until they are not required by future cells. To overcome this, without losing the generality of the model, we propose that the user should be able define how many previous iterations of the wave front should be stored in memory, and remain accessible.

In Fig. 3.2a we demonstrate this concept in action. In the first example in Fig. 3.2b we demonstrate a problem that has very simple dependencies only to the previous diagonal iteration, and the one previous to this. This example actually matches the dependencies of the longest common sub-sequence problem. Then in the second example we show a different problem which has different dependencies, and as a consequence of this requires



(a) An example of a cell in the scoring grid only having data dependencies to the previous two iterations of the wave front.

(b) An example of a cell in the scoring grid having more complex dependencies requiring the previous four iterations of the wave front.

Fig. 3.2 Different problems require different previous diagonal iterations to be stored, due to the dependencies. Dark grey cells denote the current iteration, lighter grey cells previous iterations, arrows the dependencies.

more previous diagonal iterations to be available to it. These diagrams serve to show that by allowing the user to dictate how many previous iterations are available to the current iteration, different problems with different dependency structures can be solved.

Naturally there will be some problems which will be less suitable for implementation using a paradigm such as this. For example in some problems, cells within the scoring grid are dependent on every other cell of the scoring grid, or dependent on vast swathes of the grid. These problem instances will be more challenging to adapt into a form which supports implementation through this paradigm, but again, this is linked to the point of trade offs need to be accepted when aiming for a wide reaching implementation. Later in the thesis we will consider defining problem types which are considered suitable, and problem types which are considered unsuitable.

In addition to the different dependencies, each algorithm will obviously have a different case statement or equation at its core, defining how the value for each cell is calculated.

Changing this is what allows different dynamic programming based algorithms to be implemented through the model. Therefore to allow the implementation of multiple problems we propose the user should be able to define this equation or 'core of the dynamic programming algorithm', and this will be input to the model through an input file which is discussed in detail in Sec. 3.1.3. Now we have considered how to allow the implementation of both differing dependency structures, and different dynamic programming definitions, the model should allow the implementation of different problems.

Throughout the discussion of the development of this model, we have discussed the amount of previous diagonal iterations available in memory at a given iteration. We are developing this model on the assumption that problem instances will be too large to allow the entire scoring grid to be stored in the memory of the GPU, therefore memory management will be required to take place passing partial sections of the scoring grid to the GPU at a time. Therefore, when this is being designed, the principle consideration must be placed on ensuring that the correct dependencies are available, and therefore the correct number of previous iterations are always available in GPU memory. This is discussed in more detail in the following section.

3.1.2 Memory Structure

As aforementioned, managing memory on the GPU using CUDA is a challenge due to the restrictive amount of high speed memory stores, with performance considerably dropping off as the larger global memory stores on the GPU are used. Due to this, careful consideration must be made to minimise the amount of memory required, the number of memory requests, and the acceleration of memory transactions.

When dealing with very large problem instances, naturally the size of the scoring grid will grow quickly as outlined in the problem definitions (commonly $n \cdot m$), which can become too large to fit in any of GPU memory stores. However, the problems that will benefit the most from parallelism are obviously very sizeable instances, and thus it is critical we limit the memory complexity of the algorithm to allow the computation of large problem instances.

Firstly, as introduced in the previous section, diagonal iterations that have completed can be required to satisfy the dependencies of future cells, therefore must remain available in memory. However, as discussed, the user defines the number of previous iterations that must remain in memory, therefore any iterations older than this can be safely removed from the GPU. We propose that the entire dynamic programming scoring grid be stored within the host memory, with the GPU only storing the current diagonal iteration being processed, as well as the minimum number of previous iterations required to satisfy future dependencies. This is based on the assumption that the host will have a considerably larger available store of system memory, which even in a modest high performance computing deployment is a safe assumption to make. Figure 3.3 shows an example of the transfer of data asynchronously from the GPU back to the host. The wave front and dependencies are currently being stored on the GPU, the data that is no longer needed on the GPU is being asynchronously transferred back to the host. The remaining cells to be processed are memory that has not yet been allocated on the GPU.

This means that the vast memory requirements of the GPU storing the entire scoring grid is reduced to GPU merely storing several vectors. The memory complexity of the portion of the scoring grid that is required to be stored on the GPU is now dictated by the dependency structure of the the problem which is being implemented, rather than the algorithm itself. The memory requirements on the host, rather then the GPU, remain the same as the original dynamic programming implementation, as this is still required to store the entire dynamic programming grid. Taking the example of an algorithm which requires a scoring grid of size $(n.m)$, assuming the host has 64GB of memory, the maximum dimensions of the grid which contained a standard integer data type of 4 bytes, would be roughly $63,500^2$, which in the case of some tests problems is not especially large.

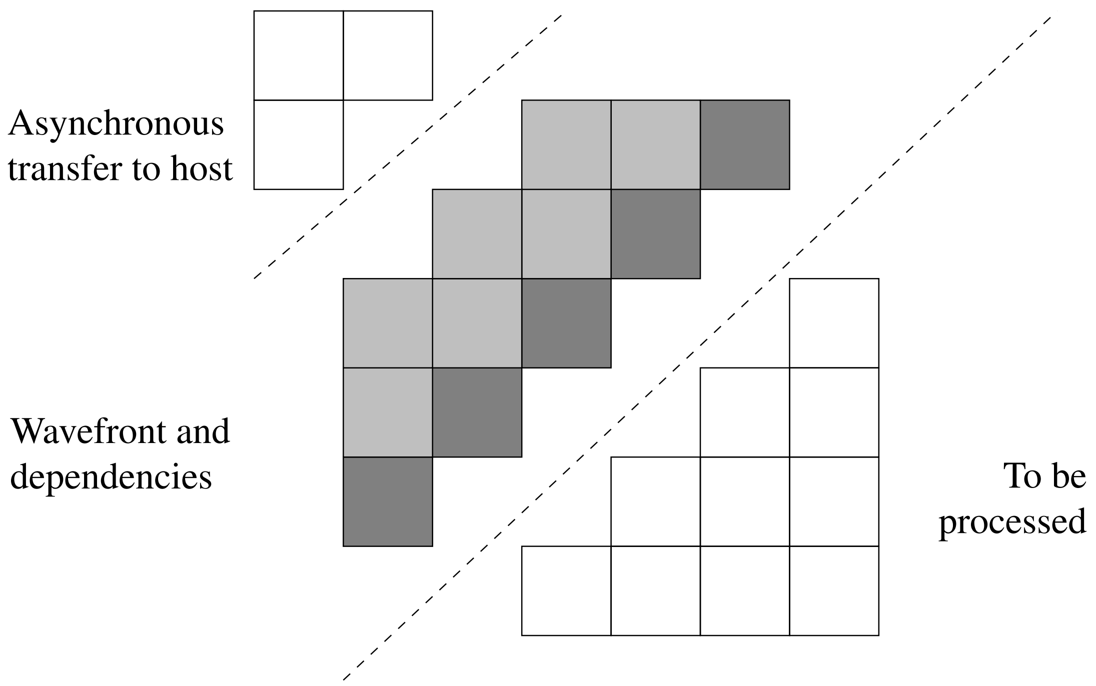


Fig. 3.3 Iterations can be transferred back from the GPU to the host when they are no longer required

To overcome this limitation we propose to allow the user to dictate whether or not they require the whole scoring grid to be available at the end of execution, or whether the solution presented by the final iterations of the scoring grid is adequate. For example in the case of the longest common subsequence problem, by considering the data from the final iteration, it is possible to retrieve the length of the subsequence. However, it is not possible to reconstruct the actual data of the sequence as this would require the entire scoring grid. With the other introduced problems, similar concepts apply, as well as with dynamic programming problems more generally. Should the user decide not to maintain the scoring grid, then there is no memory requirement on the host at all, and only the minimal memory requirement on the GPU, allowing very large problem instances to be solved. Should the user decide the entire scoring grid is in fact required, obviously the memory complexity can be a limiting factor. This is why we believe that such a decision should be left in the hands of the user, rather than being hard coded into the model.

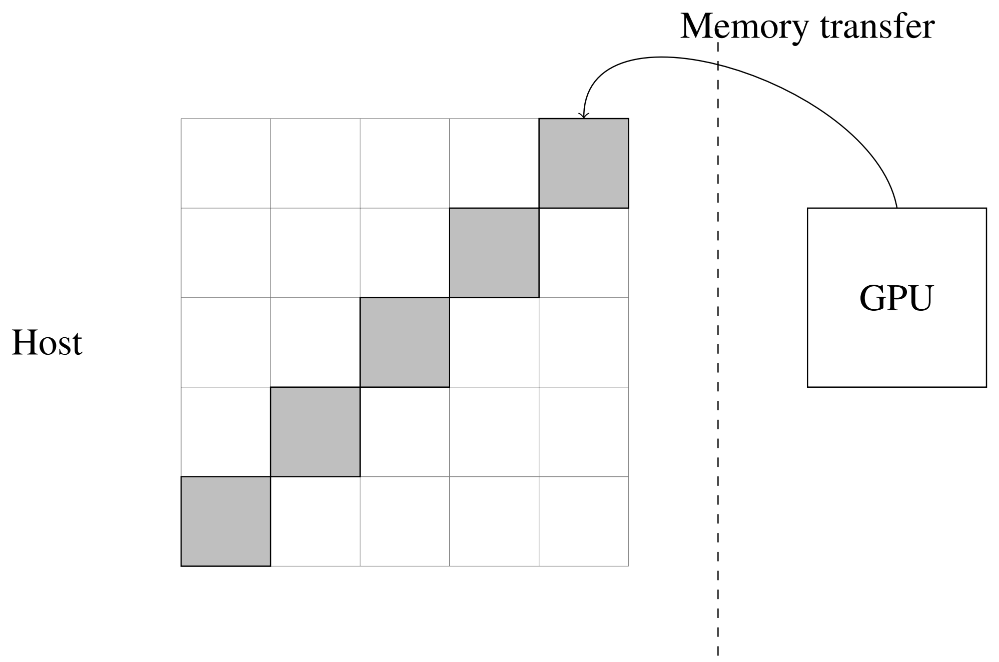


Fig. 3.4 Iterations of the wave front are transferred from the GPU to the complete scoring grid which is maintained on the host

Figure 3.4 shows these previous iterations are stored on the host to rebuild the complete scoring grid, should the user request this.

Discussed in the introduction to CUDA in section 2.4.2, is the fact that memory transfers from the host to GPU, and vice-versa, are extremely slow and often cause a bottlenecks. Therefore as we propose, if the user wishes to store the entire scoring grid, that data is transferred back to the host continuously we need a way to overcome this. Our proposal to alleviate this bottleneck is to use modern (5.5+) CUDA features, namely the ability to transfer data asynchronously to and from the host whilst computation continues on the GPU. Through the use of these features our model can transfer data from the GPU back to the host, when iterations are complete without the calculation requiring to pause. This method of memory management is likely to add some complexity to the implementation, as it will require more sophisticated synchronisation of the associated data structures. However, we believe that memory transfers are so slow this is an issue that must be addressed in our model. The complexities associated with this will be considered in the implementation section following shortly.

An alternative solution to the management of the memory which was considered at this point was to store all previous iterations on the host, further reducing memory complexity on the GPU and either transfer past iterations back to the GPU as they were required, or pass back to the GPU individual cells as needed. However considering modern GPUs have several gigabytes of memory or more, and an iteration of the wave front is only a vector of cells, it is safe to assume memory size on the GPU is no longer a limiting factor of this algorithm. Therefore, adding the increased implementation complexity, and increased run-time, of transferring data backwards and forwards between the host and the GPU, will not be beneficial to overall performance of the model.

Thus far we have identified which pieces of the scoring grid are stored on the GPU and host, and at which points of execution. Considering the data stored on the GPU, we have not yet discussed which memory store of the GPU the iteration being processed, and past iterations are to be stored in. At this point of the design process, we propose that this should be stored in the global memory store of the GPU. Whilst this is not the highest performance memory store, when considering all factors, in this specific implementation, it may prove to be the most suitable. Considering that data needs to be transferred back to the host regularly, even if the data was processed in a faster memory store, it must pass back through global memory to return to the host. This additional step in the memory transfer process is likely to add considerable additional run-time to the algorithm. The next fastest memory store, which is local memory, is limited to the CUDA block size, and CUDA block boundaries. Therefore, it could be possible the data dependency that a cell is requesting may in fact be stored beyond its block boundary, and it may not be possible to access it. Whilst there are techniques to overcome this, this is not trivial to achieve whilst keeping the model generic and suitable for multiple problem implementations. Based on all of these factors, we believe it best if the wave front remains stored in global memory, and we discuss any potential optimisations later in the implementation section.

The final point to consider is any data the problem requires in addition to the scoring grid. For example in the case of the knapsack problem, there are the profit and weight values to store, and in the case of the longest common subsequence problem there is the input strings to store. Any test data such as this will need to be stored on the GPU, as it will be required for the core calculation when it comes to filling any cells of the wave front. Based on the documentation regarding each memory store, we suggest that this be stored in constant memory. As this data is not going to be changed at any point during execution, and is going to be required by all threads at some point, constant memory is optimised at a device level to store such data. It is also possible that storing test data in texture memory may be beneficial as this is optimised for memory requests that are spatially close, and some problems use test data where the required entries are spatially close. Again, this is likely to vary from problem to problem and the highest performing solution can only be found through testing by an end user. Therefore, we propose that our model has a switch which allows the user to decide whether or not their test data is stored in the constant or texture store, based on their own needs and testing. However, this switch will default to a value of constant memory if left unset.

3.1.3 File Format

Based on the above design choices, a file format needs to be defined to allow the user to create an input file which is read by the algorithm before execution. This allows the model to be generic by enabling the implementation of new problems. As observed so far, the key factors that differ between the test problems, and therefore the factors that need to be defined in the input file are:

- The size of the dynamic programming scoring grid in use. This is both problem dependent and instance dependent, so a user must have the ability to define the size based on their specific use case.
- The number of past iterations that must be maintained to satisfy the dependencies that future cells need from GPU memory as the algorithm iterates. This factor is primarily dependent on the problem to be solved. Any older diagonals beyond this number, the algorithm will seek to safely transfer back to the system memory.
- The type of input data used for the problem. Obviously the input data can change between problems, for example it is numeric in the case of the Manhattan distance problem, however it is a series of characters in the case of the longest common subsequence problem.
- Dimensions of the input data. For example if the data is a string it would be one dimensional input data, or if it was a matrix it would be two dimensional – Therefore to keep the model generic the user needs to be able to define the dimensions of each piece of input data.
- The number of distinct pieces of input data. We need to ensure the model can support multiple input values for the problem, such a two strings in the case of the longest common subsequence problem, or single adjacency matrix in the case of the Manhattan distance problem.
- As aforementioned, which memory store on the GPU to place the test data in. The user can define whether to store the test data in the constant or the texture memory stores of the GPU based on which will give them higher performance memory access to the test data.

- A switch to define whether or not the entire scoring grid is to be maintained on the host, or whether the only data to maintain is the live wavefront that is currently iterating on the GPU.
- Arguably the most important part the user needs to define is the case definition which forms the central part of the dynamic programming algorithm and the basis from which each cell of the scoring grid is filled. This is done by writing a small function in C that interacts with the test data that the user has previously defined.

3.2 Implementation

In this section we consider the technical and practical considerations of implementing the proposed model within a CUDA based environment.

3.2.1 Thread Model

Firstly, we consider the implementation of the thread model where each thread is responsible for calculating a cell of the wave-front. As CUDA natively adopts a grid based, data parallel method, threads pass through the grid, processing cells or groups of cells at a time. This was our primary motivation for adopting a parallel method which was based around a grid based structure. However, passing through the grid in a diagonal manner adds an element of complexity and makes the implementation considerably more challenging.

Figure 3.5 demonstrates, based on the design presented thus far, how the mapping of the threads to the grid would be expected to occur. In this figure, the example problem being solved requires a 5x5 scoring grid, where the size of the largest iteration is 5, and it would require 9 iterations to solve. The values in each cell represent which iteration the cell would be populated in. If the threads were implemented in a manner such as this, it would lead to several issues. Firstly, consider the initial thread to execute, i.e. the thread of the first

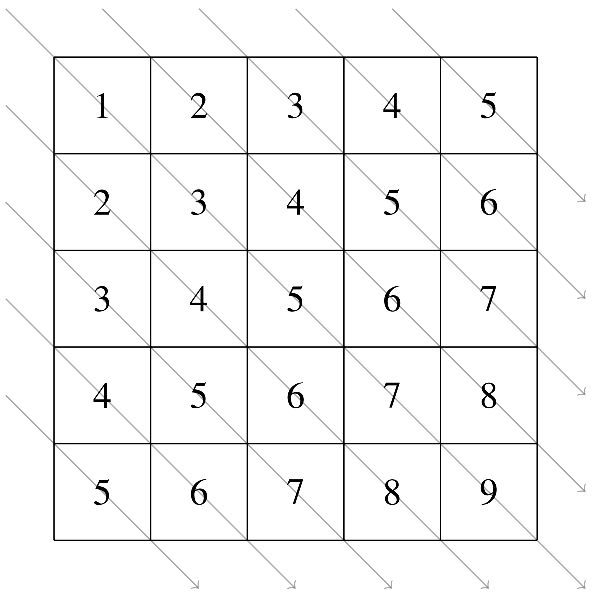


Fig. 3.5 A naive thread traversal method of the proposed model based on the design thus far, where the values in each cell represent which iteration they will be populated in.

iteration. This becomes the central thread as execution continues, running down the centre of the wavefront. As the size of the wavefront changes, this thread needs to be in a different position within the overall thread group, for example it will move from being the first thread in the group in the first iteration, to the second thread in the second iteration and so on. This means there will be a considerable amount of extra calculation each time a thread group is launched in order to have threads pass through the grid in this manner.

We therefore propose that the threads iterate across the grid from top to bottom, filling cells of the wave front as they become available. This paradigm is detailed shown in Fig. 3.6. The numbers in the cells denote the iteration of the wave front which this cell belongs to, and the grey lines passing through shows the threads moving across the data. The example problem used in this figure is the same as the previous. However this model presents its own implementation complexities – consider how thread one calculates the first iteration, then thread one and two calculate the second iteration and so on. This essentially means that thread two would be forced to idle whilst waiting for thread one to complete it's execution of iteration one, causing implementation issues, but more importantly, thread divergence.

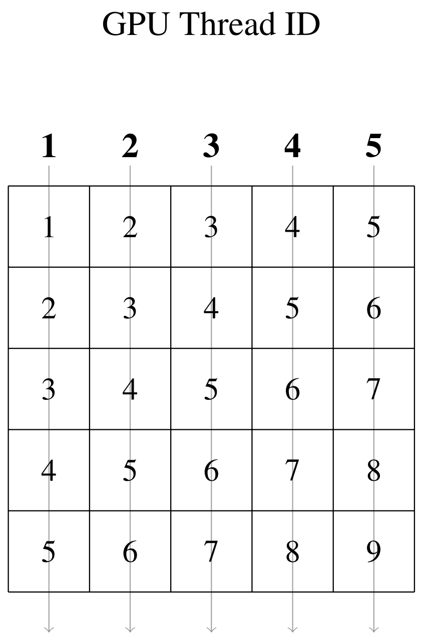


Fig. 3.6 The actual thread mapping we adopt. Numbers in each cell denote the iteration number of the wave front, and grey arrows show which threads are responsible for calculating which cell of the scoring grid

Recall that in CUDA, threads are dispatched for execution in groups called warps, where all threads are required to follow the same code path. This can lead to a large performance impact as threads stall waiting for other threads following different code paths to complete execution. Due to this, it is imperative that all threads follow the same path. Our proposed solution is to allocate memory for all threads from the beginning of execution, which allows all threads to perform work, even if this work is redundant. Following on from our earlier example, thread two will perform work during iteration one, following the same code path as thread one. However thread two will write the result of this to it’s allocated memory, which will simply never be used, and then overwritten with real data in the next iteration. Whilst this may seem as a wasteful use of cycles, performing work that is not going to contribute to the algorithm, it is more efficient than the performance impact from allowing the threads to diverge and halt waiting for one another.

In order to effectively describe this, we now present the supporting memory model, and associated diagrams.

3.2.2 Memory

The memory model we have designed to support this is shown in Fig. 3.7. This figure shows the state of memory at iteration 5 when solving the problem presented in Figures 3.5 and 3.6. In our model we allocate memory on the GPU where the x dimension (the width) of this grid is equivalent to the maximum size that an iteration could be at any point during execution – the maximum size of the wave front. In our example this means the width of the memory allocated on the GPU must be the same size as iteration five, as this is the largest the wave front can grow to. The y dimension of this matrix is equivalent to the number of previous iterations that are required to be stored, with the addition of an extra row for the current iteration being processed.

This allows all threads to perform exactly the same work, and the algorithm is not slowed down by thread groups being required to travel down different code paths. Again, using the example provided in Fig. 3.7 - in iteration three, three threads are actively calculating data, and two threads are performing the same operations, but just on unused allocated data that can be safely manipulated. Then, when the algorithm moves into iteration four, four threads are active calculating cells, with one thread performing redundant operations. Finally, as iteration five is the largest iteration of execution, all threads are now active and have work to perform. From this point forward the process continues in reverse.

Recall from our introduction to CUDA, the structure of memory transactions. If memory requests take place from sequential addresses, CUDA will coalesce these requests, and multiple requests will be served in a single memory transaction. This single coalesced transaction can be as large as an entire warp, assuming that all the memory accesses are totally sequential in memory. Therefore by rotating the diagonal structure of the parallel

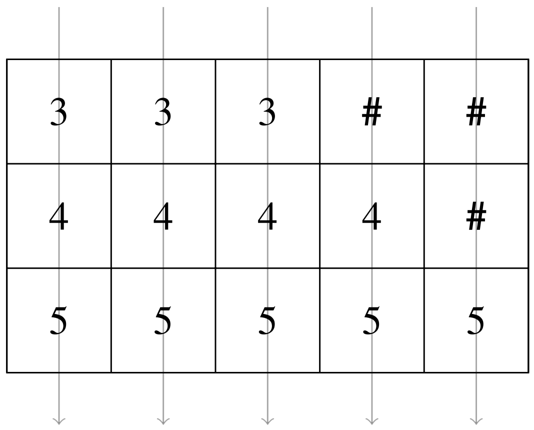


Fig. 3.7 The memory model which supports the parallel model. Numbers within the cell denote the iteration of wavefront the cell belongs to, '#' denotes the memory has been allocated, but is not currently being used by the model.

model, and placing it linearly in memory, not only can we overcome divergence issues, but we can also ensure that memory transactions are being performed efficiently. This further validates our earlier design decision of allocating memory for each thread, even should this data be totally unused, as it allows the number of memory transactions to be minimised.

Now we have considered the structure of the memory, and how this will be stored, next we will discuss in more detail the removal of previous iterations from the GPU that are no longer needed. Regardless of whether the user wants to store the entire scoring grid on the host, or just maintain the wave front on the GPU, in both cases the memory will need to be rotated as the wave front iterates through the scoring grid, as one of the major design principles of this model is that the issue of limited memory resources on the GPU is overcome. We use the term 'rotation' in relation to the memory model, as the solution we propose is all rows stored in GPU memory are moved up by one at the end of each iteration, and the oldest row that is no longer needed is transferred asynchronously back to the host if this is what the user requests, or overwritten if not. An example of this is provided in Fig. 3.8.

This diagram continues from the earlier examples where the largest iteration of the wavefront has a width of five, and only two previous iterations are required to be stored. The set of cells on the left shows the state of the memory at iteration five, and the set of cells on

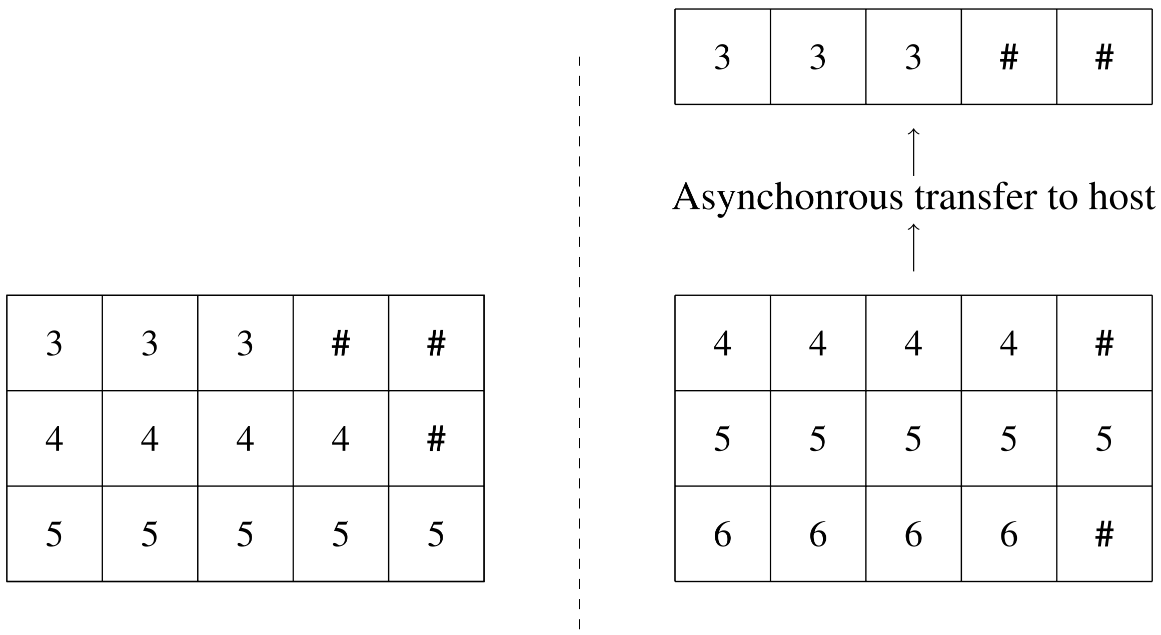


Fig. 3.8 The memory model which supports the parallel model, showing that at all times, threads have work, and memory allocated to perform this work in. Numbers within the cell denote the iteration of wavefront the cell belongs to

the right the memory at iteration six. The oldest iteration, which at this point is iteration three is passed back to the host asynchronously if the user has opted to maintain the full scoring grid. Through this method it means that only the minimum amount of data is stored on the GPU. However it does introduce the issue that synchronisation is required between iterations of the wave front to allow memory rotation to occur, and ensuring threads are not rotatin memory early causing race conditions. Synchronisation is covered in the following section, 3.2.3.

Considering next the details of the asynchronous transfer. Modern implementations of CUDA allow computation and memory transfer operations to take place simultaneously, through the use of a mechanism called streams. Streams are a method of dispatching work to the GPU – each have their own queue, and are executed simultaneously. Therefore by queuing memory requests in one stream, and queuing kernel executions in another, both can take place at the same time. This requires additional care be taken to ensure that kernels are not requesting data that is not yet present on the device, and vice-versa.

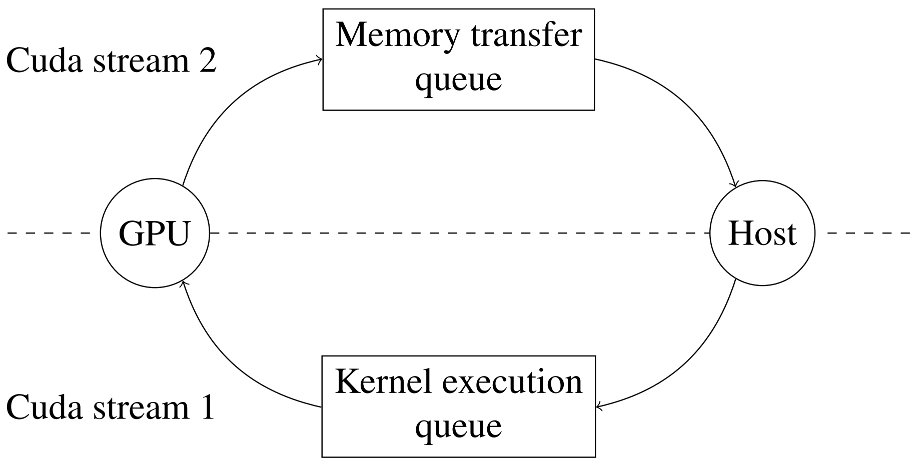


Fig. 3.9 Multiple CUDA streams are used to allow the asynchronous operations of data transfer and processing

We initialise and allocate the memory on the GPU in the standard manner, as well as allocating the data on the host if required. Then, when execution begins, kernel commands are sent to the GPU in one stream, whilst the GPU itself queues old iterations for return to the host in a separate stream used for data transfer. As adding elements to a CUDA stream is a non blocking operation, therefore in the implementation of our model we can simply queue all kernel commands into the kernel stream at the beginning of the algorithm execution. Then the host can either continually check for incoming data from the GPU in the data stream, or if the scoring grid is not being maintained on the host, simply wait for kernel execution to finish and return the final cell of the dynamic programming scoring grid. An example of this in use is given in Fig 3.9

At this point it is possible to identify a potential flaw in the proposed memory structure and memory management model. As data to be transferred back to the host resides in the queue of the CUDA stream on the device until it is transferred back, it is possible to be in a situation where the queue is filling at a faster rate than the calculation is taking place. In this scenario it would be possible that the queue can grow to such a size that memory runs out on the GPU, causing the model to crash. This problem is also compounded as memory transfers are so slow, it would be expected that the wavefront would be iterating through the grid at a considerably faster rate than old iterations can be removed. However the extent of this problem will be largely based on the complexity of the operation each cell must perform to fill its data, for example, if the wave front is performing very complex operations to fill the cell of the grid, it may infact traverse the scoring grid slower than the queue is being emptied. In section 3.2.4 we perform some small scale testing to identify the scale of this problem, and to validate our design choices.

The other potential problem of this method, is that at the end of execution there will still be a pause whilst the remaining data transfer to the host has to take place. This is an unavoidable side effect of the model we are proposing. However, as we are using asynchronous transfer throughout, we are still in a beneficial position compared to standard data transfer. If standard transfer was being used it would be a case of execution, then a pause as transfer back to the host takes place, then execution again. Although there is still likely to be a delay at the end of execution within our model, this has been offset greatly by the fact the majority of transfers have been overlapped with execution.

3.2.3 Thread Model II

Now the supporting memory model has been introduced, we can address the finer points of the thread model.

When CUDA was first introduced in section 2.4.2, we discussed the concepts of blocks, which is a method of decomposing the data. As we are now proposing to launch a number of threads equal to the biggest iteration of the wave front in an effort to overcome thread divergence, we are now requesting more threads to execute than the device can provide. Therefore smaller blocks of data will be processed at a time, and the blocks will be processed by smaller groups of threads as the resource become available on the GPU. We must therefore consider the block decomposition of the memory that is being processed by threads on the GPU.

Ideally, we want to achieve the optimal mapping of blocks to the device resources. However this is dependent on factors such as the amount of memory in use, and even the amount of registers in use on the GPU. Therefore, at this point of developing the model we cannot consider such factors, as they will be highly problem specific. The only optimisation we are able to make from this high-level of the design is to ensure that the block sizes are a multiple of the size of a warp, allowing efficient mapping of threads to data. In the following performance validation section, 3.2.4, we ensure that common block sizes do in fact perform efficiently using the model, and demonstrate this is a valid assumption to make.

At this point the structure of our data follows a simple, regular structure. The division of the scoring grid into blocks occurs thus; decomposition of the scoring grid into the appropriate block size, ensuring it is padded to both the size of the longest iteration, and padding it further to a multiple of the block size if required. Mathematically the number of blocks in use would be denoted as $\lceil \frac{n}{b} \rceil$, where n is the size of the largest iteration of the wave front, and b is the desired block size. We ensure $b \mod t = 0$ where t is the size the size of a thread group executed by the device. Figure 3.10 shows an example of this in action where the size of the wave front is five (n), and the block size is four (b). For the purposes of this example we assume the block size of four is a multiple of the size of the thread group (t), however in reality this is unlikely.

In our efforts to overcome the issue of thread divergence, it is possible we have introduced new performance issues into the model. Consider the case, for example, of test data which has millions of elements, causing the longest iteration of the wave front to be several million in length. This means that during the first iteration for example, there can be one thread running on useful data, then other millions are simply wasting cycles. As a GPU does not have this many threads, it is a case of loading and unloading blocks such that all cells are filled, and this only compounds the problem of wasted work further, as this means blocks are being spawned for the sole purpose of redundant work. Figure 3.11 demonstrates this

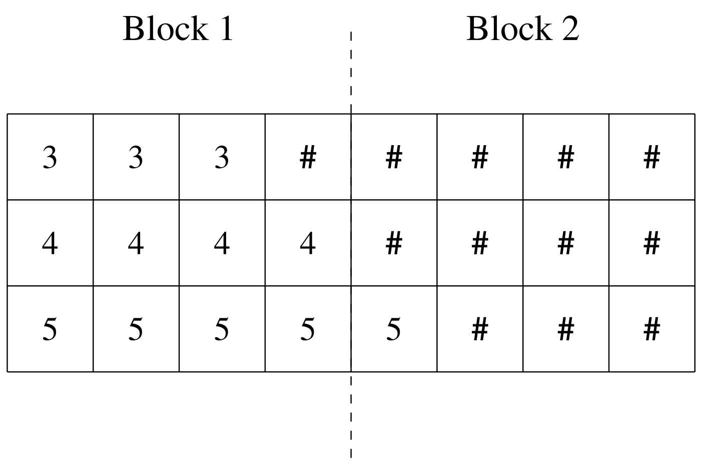


Fig. 3.10 Block division where the block size is four, and the size of the largest iteration of the wave front is five

concept. We proposed this padding solution as a method of overcoming divergence, so that thread blocks are not being wasted, but in this scenario we now find we are launching blocks for the sole purpose of wasting them.

We therefore require a method to overcome this issue. As it is trivial to determine the size of the current iteration, by simply using using the iteration number, we can identify which blocks are going to have no work at all. We can then launch the CUDA kernel for that iteration without the additional blocks.

As mentioned earlier, during the memory rotation operation, we need to ensure synchronisation of the threads. However, synchronisation is against the design paradigm of CUDA as a data parallel language. Consider the situation where as many blocks are being processed as possible, but all of these are waiting for future blocks to complete. Due to the CUDA policy of once execution has started on a block it will remain loaded until it completes, the model will now hang as deadlock has been reached. Due to this CUDA provides API features that allow threads within the same block to be synchronised, but does not provide any functionality for synchronisation across block boundaries.

It is expected when developing CUDA applications, and data wide synchronisation is required, this is achieved by launching a new kernel. This means that when one kernel

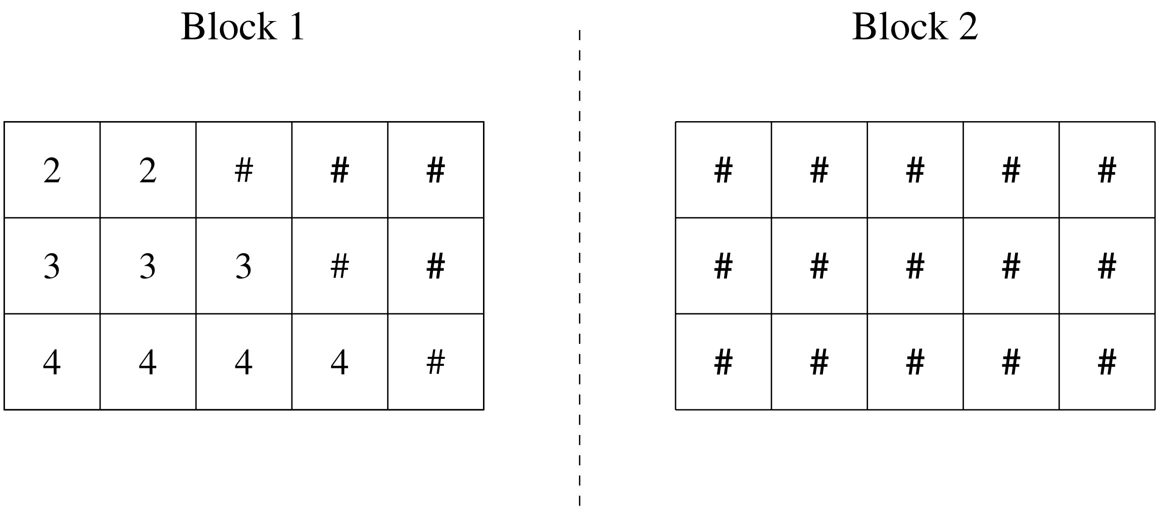


Fig. 3.11 The memory used when the longest iteration is 10 wide, spread across two blocks

completes, it guarantees that all threads are at a given point, and all data is in a given state. Then a new kernel can be launched to continue with processing. In our scenario this means that a kernel can only compute one iteration of the wave front, as we require synchronisation after each iteration. This may sound like a sub-optimal operation, however the process of launching a kernel is a very quick, near instant, process, and the CUDA documentation describes this as the only way of achieving a safe global synchronisation point.

We can accelerate this process slightly, by queuing all the kernels required through execution within the CUDA stream responsible for dispatching kernels, meaning there is minimal input from the host between kernels. This also is beneficial for the model in a more significant way however; as we have a different kernel for each iteration, it allows the model to alter then number of blocks being launched as well as the number of threads, for each iteration, allowing us to prevent the launching of blocks with no work.

3.2.4 Performance Validation

We now perform some small scale tests, to validate the claims made in previous sections, and justify our design choices.

Firstly, we consider the issue of the asynchronous transfer queue, to evaluate whether it is possible for the wave front to be calculated quicker than the GPU can transfer completed iterations off the device, potentially causing the model to crash. For this test, we assumed each iteration consisted of 500 cells, and each iteration when completed transferred back to the same location on the host, i.e. overwriting, so that memory constraints on the host would not limit the size of the test. This allowed us to run an artificially large number of iterations to provide the best chance of overfilling the queue if this was a weakness in the model.

The test was run at three different workloads - the low workload denotes simply loading the current cell from the wave front, one other as a dependency, and finally writing the same value back incremented by one. The medium workload is defined as the same, but performing 10 GPU math based operations, `max`, `abs`, `min`, in between the load and the store. Finally the high workload performs 30 GPU math based operations. For all testing it was assumed that there was only one previous diagonal to store on the GPU. Results of this test can be found in Fig. 3.12.

The results show that all tests successfully passed without crashing, however in the case of 10^8 iterations, at a low workload, the transfer queue was very close to the limit of the 6GB of memory available on the GPU. However, this test still passed in an artificially constructed worst case which is very promising. When using an iteration count this high, if the data from the GPU was actually required to be stored on the host, large amounts of host memory would be required, and if it was not the asynchronous transfer queue would not be needed – therefore this is a good representation of an worst case for the model.

Next we briefly consider the effectiveness of padding the data to ensure that divergence is minimised. This is a very common paradigm in CUDA and a recommended design pattern, but it is still prudent to validate our claim, as in the early and late iterations the amount of padding required can be high as size of the wave front is small, and the block size is large.

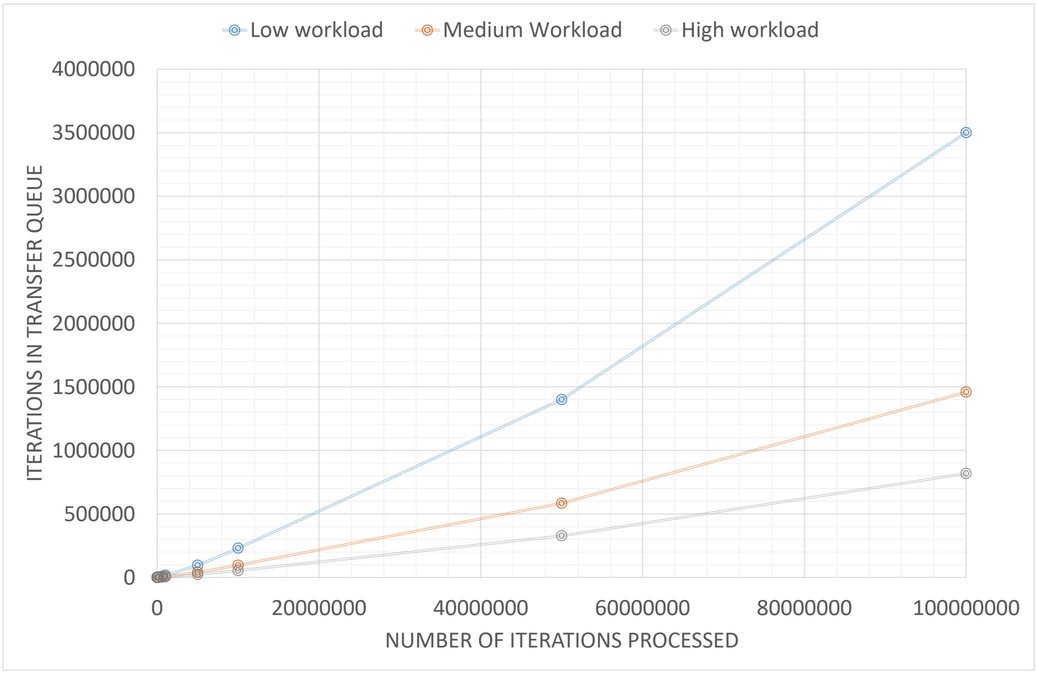


Fig. 3.12 Number of iterations still in the asynchronous transfer queue at the end of execution for differing workloads of wave front calculation

In this test we consider two factors when analysing whether the padding is beneficial. Firstly the run time is recorded to see the effect padding has practically on the run time of the algorithm, and secondly we record the CUDA metric of *branch efficiency* which records at a hardware level the divergence of a given program. Full details of this metric are provided later in Section 5.2.1.

To test this we create two instances; one where padding was not present, and one where padding was. We ran tests where blocks were 'full' to different percentages, i.e. the amount that would have to be padded, or would be excluded by an if clause if it was not in use. The size of the wave front was set at 4,096 elements and the size of a computation block at 4,096. We do not need to consider multiple blocks, as we have already determined that the algorithm will be intelligent enough to not spawn blocks with no work at all. We ran the high workload test case from the asynchronous transfer test for 1,000 iterations to identify the average performance impact of using padding.

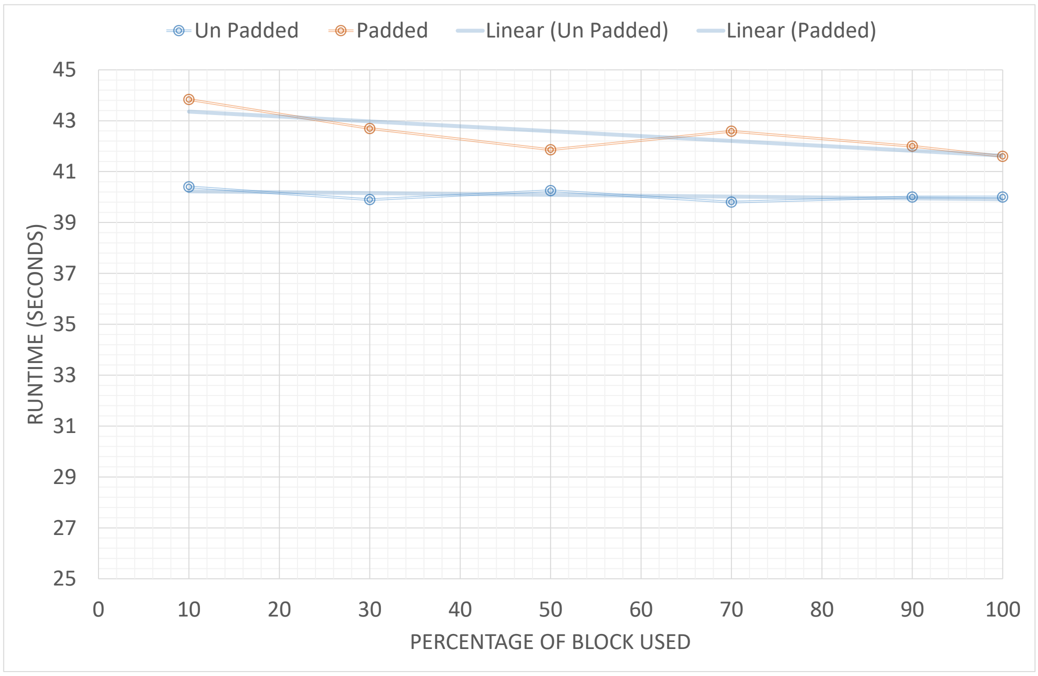


Fig. 3.13 The effect that padding unused data has on the runtime of algorithm, as the amount of unused data in a given block decreases

The results of this test are shown in Fig. 3.13, and they support our statement that providing threads with an area to read and write to so they can follow the same code path as other threads, improves the overall runtime of the algorithm even though more work is being carried out. Note from the graph that performance improves for the unpadded version at the 50% mark before deteriorating again, we discuss why we believe this is shortly with the divergence results. Replicate runs of the test were carried out to ensure the validity of this result. Another surprising point of test was the lines did not converge to the same run time when the block was 100% used, when it is expected both the padded and unpadded implementations to be following the same code path. We conclude from this result that the `if` clause that guards the out of bounds elements in the unpadded implementation is likely adding a small amount of overhead, so even when the block is entirely used it is still performing slower than it's unpadded counterpart.

Percentage Used	Padded	Unpadded
10	1	0.95
30	1	0.86
50	1	0.99
70	1	0.85
90	1	0.97
100	1	1

Table 3.1 Results of the CUDA branch efficiency metric when running on both padded and unpaded data, as the percentage of 'used' data increases

Results of the testing for recording the branch efficiency are given in Tab. 3.1. As is expected, it shows no divergence for padded data, however the unpadded data does not present the completely linear trend initially expected. This is due to the fact that the branch efficiency is *not* dictated by the amount of unused data, but instead dictated by the amount of warps that have to traverse different code paths. For example it would be possible to have no divergence at all, if the boundary between the code paths each warp must take lines up at the boundary of a warp size - i.e. in our example of a block size of 4096, in the 50% used data case, 2048 cells must be filled based on one code path, and 2048 on a different. Therefore if the warp size was 16 threads, the first 2048 cells would be filled by 128 warps, and the second 2048 by another 128 – however at no point did a warp need to split based on an `if` clause: all threads in every warp followed the same path, although these are different paths. However, we will be using the optimised padded version.

3.2.5 Input File Implementation

Based on the requirements of the file format identified in Section 3.1.3 we consider each of these points individually. For maximum compatibility across systems and inter-operability, we assume the input file is simply a plain text file, with each new line defining the next input parameter to be processed. Each line of this input file is read, and parsed by a Backus-Naur Form (BNF) based syntax parser, and loaded into the model. Please note for brevity we

```
5x5
prev: 2
char 1x5 A,B,C,D,E
char 1x5 C,D,E,F,G
mem: 0
maintain: 1
```

Fig. 3.14 A complete input file for a longest common subsequence problem, with two input strings 5 characters in length, stored in constant memory on the GPU, maintaining the entire scoring grid

omit the BNF definitions for standard text forms such as ‘character’, ‘integer’ and ‘line terminator’.

Allowing the user to define the size of the scoring grid is a case of having it defined in the input file, as two integers which represent the *n* and *m*. When the file is read, the model is then aware of how much memory to allocate on the host device to maintain the entire scoring grid. Also from these dimensions, the algorithm can calculate the size of the longest diagonal, so the width of the grid that is to be maintained on the GPU can also be calculated. The input file should contain two integers on a single line, separated by the character ’x’, allowing it to be captured by the respective BNF tokens `<dimension> ::= <n> "x" <m> <EOL>`, `n ::= <integer>`, `m ::= <integer>`.

The second value that is defined in the input file is the number of previous diagonals to store in memory on the GPU. This is defined by the user based on the dependencies that are present in the dynamic programming definition. In terms of the memory that is stored on the GPU during execution, this value represents the height of the grid the GPU is maintaining in global memory. As introduced in Sec. 3.1.2, once rows are rotated past the top of this grid, they are asynchronously transferred back to the larger grid being maintained in the host memory, should the user require. This is represented in the input file by an integer prefaced by the characters `prev:` allowing it to be captured by the BNF token `<previous> ::= "prev:" <integer>`.

Now that the dimensions of the scoring grid to be maintained in both system and GPU memory are defined, the user needs a way of passing in the input data which the algorithm is to process. Each piece of complete input data must be on a single line of the input file. On each line first the user needs to define the type of this specific piece of data. For this the line must start with a small string token identifying the type, such as '`char`', which would represent input data that is in the format of characters, '`int`', which would denote the input data is going to be a sequence of integers, and so on. Following this, the user needs to define the dimension of the input data. Finally, following this, the user only needs to provide a string of comma separated values which form the actual test data, with escape characters denoting when each row of the test data matrix ends, should it be 2D. The user can add as many lines of input data as they wish to the file which will be captured by the tokens:

- `<input> ::= <type> <dimension> <input-data>`
- `<input-data> ::= <data> <EOL> | <data> <data>`
- `<data> ::= <data-row> <EOL> | <data-row> "&" <data-row>`
- `<data-row> ::= <comma-sep-values>`
- `<comma-sep-values> ::=` (omitted for brevity)

For example to represent a string for the longest common subsequence in the input file a user would write `char 1x5 A,B,C,D,E`. After defining the dimensions of the scoring grid, and the number of diagonals to maintain, the user can define as many lines of input test data as they wish, all of which will be accessible from model.

Finally the user places on the last lines of the input file a switch, denoting whether test data should be stored within the texture memory or constant memory. We assume constant to be the default option therefore setting this switch to 0 uses constant, and 1 uses texture. Also the user should define whether they want to maintain the scoring grid on the host as well, or just the current iterations on the GPU. This defaults to only storing what is currently on the GPU, therefore a value of 0 does not maintain the whole scoring grid, and a value of 1 does. These will be captured by the BNF tokens `<mem> ::= "mem: 0" | "mem: 1"` and `<persist> ::= "maintain: 0" | "maintain: 1"`, respectively

A complete example of this, using the longest common sub-sequence as problem would appear as shown in Fig. 3.14

Representing the Dynamic Programming Case

Allowing the user to define the case definition of the dynamic programming case is a more challenging consideration, as it must be present at compile time for the model to be able to be able to generate the CUDA kernel. Due to the fact it must be present at compile time the definition of the dynamic programming case cannot be in the input file as with the other parameters, and must be written into a small function. To make this as easy as possible for the user, we develop small wrapper for the user, allowing them to implement different problems without needing to know the intricacies of CUDA programming.

A function is defined which is called each and every time a cell of the scoring grid is required to be filled, and the user simply needs to fill in the definition of the dynamic programming case here before compiling the program. This function is provided with a struct containing pointers to all the input data the user has defined, allowing the user to access all of the data they have specified in the input file. Also this function is provided with the current iteration number, as well as the i and j index of the cell that is being filled.

This function is also provided with the length of the current iteration, which is used during memory accesses, as well as a pointer to the data struct containing the number of previous iterations the user has opted to maintain, allowing them to load any data dependencies they need. It should be noted at this point that obviously memory access to previous iterations cannot simply take place through the desired (i, j) values. For example, if the wavefront was in the middle of the scoring grid, and needed to load a cell from the previous diagonal iteration, the actual (i, j) value of the desired cell in the context of the entire scoring could be a very large number. Therefore this cannot be used to directly load data from the smaller grid on the GPU which is maintaining previous iterations. We provide a function the user must use when making memory accesses to previous iterations, allowing the user to make access to previous diagonals through global (i, j) references. This is covered in the following Sect. 3.2.6

In this way it is hoped that the user should be able to write very basic C code to represent the core of there problem, using the most simplistic algorithmic representation, and the model will automatically handle everything else. Also, using the approach of allowing the user to provide their own code may allow for more advanced users to provide more optimised functions.

The actual source for the function signature the user is to fill to define their problem is defined as:

```
__device__
UserFunction(data_struct Data,
             unsigned int CurrentIteration,
             unsigned int i, unsigned int j,
             unsigned int Length,
             unsigned int* PreviousDiag)
```

Obviously, a user is then able to create additional device functions which are called by this core function if they so desire, and extend this functionality as far as they need too.

3.2.6 Other Implementation Details

Now that the model has been defined we consider the finer details of the implementation, which were only introduced briefly earlier in this chapter.

As the model needs to be generic we need a method which allows the user to input different types of test data, as well allowing different amounts of this input data. Therefore to enable this we use pointers to void data structures storing the raw data, which are then cast to the appropriate type based on what the user has dictated within the input file. To allow the storage of many pieces of input data, we store an array of void pointers, which can grow arbitrarily large as memory allows, where each pointer maintains a reference to a different piece of input data. It was considered at this point that a mechanism such as header guards, and the identification of types from the input file if present at compile time, could have been used to alter the data types of the input data in a more robust manner. However, it was decided that whilst the dynamic programming definition should be present at compile time, to require the input file at compile time could be inconvenient as this would lead to the need for recompilation for changes as small as altering the input test data, or changing the memory store that is being used for the test data. Therefore, it appeared that the manipulation of the raw data through void pointers allowed for a more convenient and usable final solution, although additional care must be taken when manipulating the data types. Finally, the switching between GPU memory stores is controlled by a basic `if` clause dictating where the memory should be allocated and transferred to. Note, that the input file is parsed through the use of regular expressions, allowing large input files to be scanned and read in quickly.

At the beginning of the execution the scoring grid on the host is allocated if required by the user, and the scoring grid is created on the GPU based on the number of previous iterations that are required, and the test data is transferred to the appropriate store on the GPU. Then, the two queues for kernel execution and memory transfer are both initialised.

After the kernel queue is created, before each kernel is added we perform some pre-processing in an effort to maximise efficiency, as covered briefly earlier. The width of each kernel of the wave front can be different, as a new kernel is launched for each iteration due to the fact global synchronisation is required. Therefore as the kernels are queued to be executed, the optimal width of each iteration is calculated to minimise the number of blocks, whilst ensuring that only full blocks are present in the kernel using padding where required to achieve this. The question of optimal block size is a subject of considerable research within the CUDA community, and a factor that is heavily influenced by the underlying hardware available, and the properties of the problem that is being computed such as the amount of local memory required, and register usage. Therefore, instead of having this variable within the model, we elect to hard code a block size that is optimal for the hardware that is used during testing based on the CUDA occupancy calculator [72]. This is discussed in more detail in Chapter 5, when the testing environment is outlined.

We provide a memory access function that allows the user to access data they desire from previous diagonals, using a reference based on the entire scoring grid. This wrapper must therefore be used in all accesses to previous iterations to ensure the validity of the result. The signature of this function is very simple with the user simply passing the (i, j) of the location of cell in the grid that is currently being processed and the (i, j) of the cell which they desire to access, as well as the length of the current iteration to allow offsets to be calculated accurately. It is for this reason the defined user function is provided with information regarding the length of the current iteration. Therefore, in the dynamic programming definition when the user defines read or write operations to or from the scoring grid, all memory read and writes must pass through this wrapper.

The kernel to rotate the memory is a very simple function to iterate across each row representing a previous iteration, and move it up one, starting with the oldest iteration which is passed to the queue for the transfer back to the host if the user has opted to do so. The only point of note is that the threads are locally synchronised block wise before the next row is rotated, as this allows the memory requests to be coalesced and therefore accelerated. This additional kernel is queued between each kernel execution.

The mechanics of handling the parallel nature of the execution of the algorithm, simultaneously with the memory transfer, is actually surprisingly simple thanks to the functionality provided by CUDA streams. At the beginning of execution, all kernels are queued in the execution stream as aforementioned, then the host is simply required to monitor the incoming data in the data transfer queue waiting to handle data as it arrives. Therefore from the host point of view, once all the kernels have been created, they need not be considered again and are handled by the CUDA driver.

Finally execution of the algorithm does not terminate until the queue to transfer back completed diagonals from the GPU is empty, if they are being stored on the host to reconstruct the entire grid, or until the GPU returns the most recent iteration. From this point the user is expected to separately process the outputted data to their own needs to produce a solution from algorithm output.

3.3 Comparison to Existing Work

We will now discuss the model's novelty compared to the existing literature, and what benefits the proposed approach offers.

Our approach is closely based on the method of filling individual cells of the scoring grid in parallel, using a different thread to fill each one as it's dependencies become satisfied. This in itself is not entirely novel; the works of Krusche and Tiskin [54], Kloetzli et al. [53], and to a lesser extent Boyer et al. [14] are existing literature's with similarities to our implementation, and the literature review in Sec. 2.4.3 covers these in more detail. However, as discussed at length through this thesis, none of these parallel models allow for the parallelism of other problems, quickly and easily. The above papers for example are all related solely to the solving of the longest common sub-sequence problem. Considering again the early work of Galil and Park [30], where they discuss the feasibility of a more generic diagonal approach to solving dynamic programming algorithms, you can also see clear influences for our model, as they consider dynamic programming from a problem agnostic point of view. Furthermore, from an implementation point of view, of the previous works we have identified few of these consider factors such as optimising CUDA parameters, or take advantage of modern CUDA features such as streams.

Therefore, a clear, and significant, contribution of this work, is that we demonstrate that it is possible to make a pseudo-generic framework that allows the implementation of many different dynamic programming algorithms based problems in parallel. Compared to the existing literature we reviewed, none of these allowed for the practical implementation of different problems; they either demonstrated an implementation for a single problem, or discussed the theoretical implications of multiple problems, with no practical backing. We then go a step further and implement the model upon the architecture of the GPU, demonstrating that our model is suitable for use in a massively parallel environment. Of the literature reviewed, there is considerably less available regarding implementation on GPU architectures, with there being some notable examples from post 2000 [13, 75, 99]. Therefore, not only are there are no examples of CPU based models to solve multiple problems, there are also no examples of GPU based models, allowing us to find our research niché.

Our model also employs novel memory management, which we believe has two fold contributions. Firstly, and most significantly, it allows problems larger than the size of memory to be computed through the use of memory rotation, and secondly it demonstrates a highly efficient implementation taking full advantage of multiple GPU streams, and asynchronous operations. The base memory structure initially stemmed from the work of Klotezli et al. [53], but has been vastly improved from this point with additional implementation optimisation such as memory padding, as well as being extended to enable the implementation of memory rotation.

In terms of the finer details of the implementation of our model, Wu et al. [95] also provided inspiration for our work demonstrating how the size of the wave front can be adjusted based on the amount of work that needs to be done in an effort to improve the parallel efficiency. This proved to be a critical factor for our implementation, considering the large impact divergence can have on GPU code. We also drew on the work of Berger & Galea [10] where they discuss how thread grouping and the altering GPU parameters, based on the problem at hand can improve the efficiency and performance of the algorithm. Whilst our model does not adopt the concept of thread grouping, we have a strong emphasis on allowing the user to change parameters based on the problem being computed, as well as an element of pre-processing where we calculate optimal values for block sizes, and kernel sizes.

Finally, our contribution of allowing the user to interact with the model through the concept of a file format, in an API style mechanism seems to be entirely unique in the literature, although this is arguably more an implementation and engineering contribution rather than solely a scientific contribution. However, it is through this mechanism we are able to claim our model is generic, and without the ability to define different problems, as well as associated parameters, the rest of the model would not be valid.

The complexity for our proposed model, to compare to the existing literature, can not be ascertained at this point, as this is dependent on the problem the user inputs, as well as the amount of memory they opt to maintain. Therefore the complexity for individual problems is considered in the following implementation section.

Summary

This chapter presented a detailed description of the parallel model that this thesis is proposing, as well as giving an insight into the design process to provide an insight as to our design choices. Also the novelty and contributions of our model were presented in comparison to the literature currently available. Performance results were shown to justify some of our design choices, demonstrating the general performance of different components of our model, such as memory transfer and block structure, and how these perform in isolation. The next chapter describes the implementation of the introduced test problems on the GPU, through the use of our parallel model.

Chapter 4

Application of the Model

Overview

In this chapter we discuss how the introduced problems are solved through the use of the proposed model. We consider different dynamic programming definitions, which require different dependencies to be maintained for different problems, and how this is represented in the model. We also consider implementation optimisations which can be made for the separate problems. Later in the chapter we discuss adaptations required in order to solve more complex dynamic programming problems. Finally we seek to define the broad classes of problems which are unsuitable to be solved by the model.

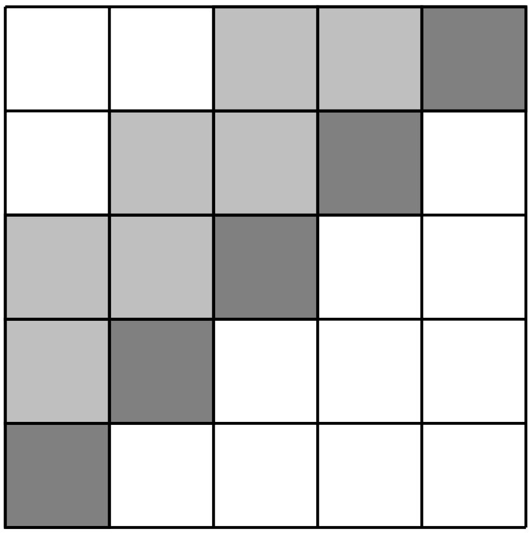


Fig. 4.1 Dependencies of the wavefront when solving the longest common subsequence problem

4.1 Problem Implementation

First, we consider how each introduced test problem is represented within the model, in terms of dependencies the current iteration of the wave front requires, as well as other details such as how test data is represented. Maximum theoretical problem instance sizes are also defined based on the code being deployed on a GPU with 6GB of memory.

4.1.1 Longest Common Subsequence Problem

The dependency structure for the longest common subsequence problem is given in Fig. 4.1, with dark grey cells denoting the wave front, and light grey cells showing the dependencies. White cells in the top left, behind the dependencies, are cells which can be transferred back to the host if required. Therefore in the case of this problem, only 3 vectors need to be stored on the GPU. The memory complexity of algorithm on the GPU is therefore $O(n)$ where n is the length of the longest input string. Due to this low memory complexity, we believe that if the scoring grid was not be stored on the host, and assuming the data type of the scoring grid was a 8 byte long integers, strings of length roughly 150,000,000 could be solved before

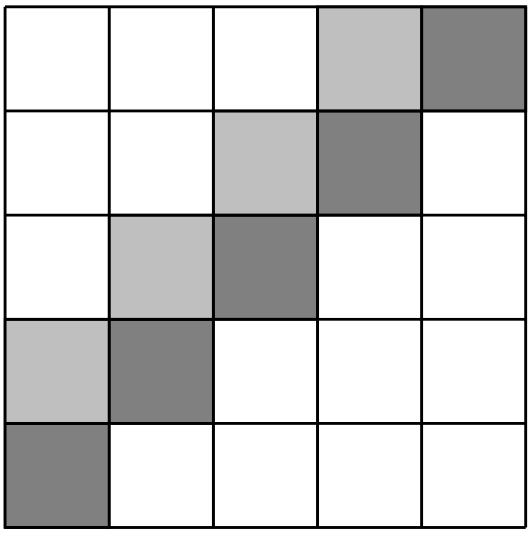


Fig. 4.2 Dependencies of the wavefront when solving the Manhattan tourist problem

exceeding the 6GB limit of our GPU hardware. This figure assumes there both test strings are stored on the device, and the scoring grid is required to store long integers rather than regular integers due to the possible value of the longest common subsequence. Finally, both input strings will be stored in constant memory on the GPU, as characters of vectors.

In the case of the edit distance problem - the dependency structure is identical to that of the LCS problem. Therefore the above approach can also be used for the edit distance problem by only changing the dynamic programming case statement.

4.1.2 The Manhattan Tourist Problem

The Manhattan tourist problem follows a similar structure to the LCS problem, but requires less previous dependencies. Figure 4.2 shows the dependency structure of this problem, again with the wave front denoted in dark grey and the dependencies in lighter grey. Due to the dependencies of this algorithm the memory complexity is still $O(n)$, where n is the width of the scoring grid. The test data requirement for this model is higher, however. We store the input test data in two matrices, where the first matrix stores the vertical edge weights, and

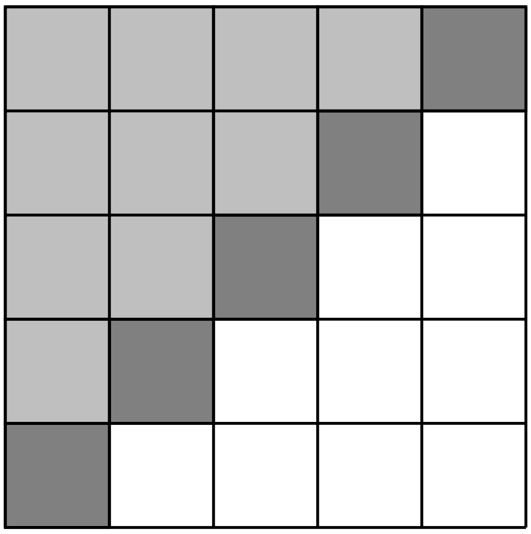


Fig. 4.3 Dependencies of the wavefront when solving the knapsack problem

the second matrix stores the horizontal edge weights. Each row of the input matrices stores a complete column, or rows worth of edge weights.

Based on these increased test data requirements, we now attempt to determine the theoretical maximum problem instance size of the model. Based on two square input matrices representing the edge weights storing 4 byte integers, and the scoring grid also storing 4 byte integers, we believe the maximum theoretical city dimension our model can solve is $25,000^2$. When running the Manhattan tourist problem, we elect to store the test data in texture memory as it is possible requests may be spatially close within the 2D space.

4.1.3 The Knapsack Problem

Next we consider the knapsack problem; compared to the previous problems, this has a more complex dependency structure. It is possible for a cell to require the entire row the cell is stored in, from column 0, through to the column the current cell is in. Therefore, a much larger number of previous iterations must be maintained to satisfy the dependencies. Until the wave front reaches the halfway point of the scoring grid, all previous iterations must be

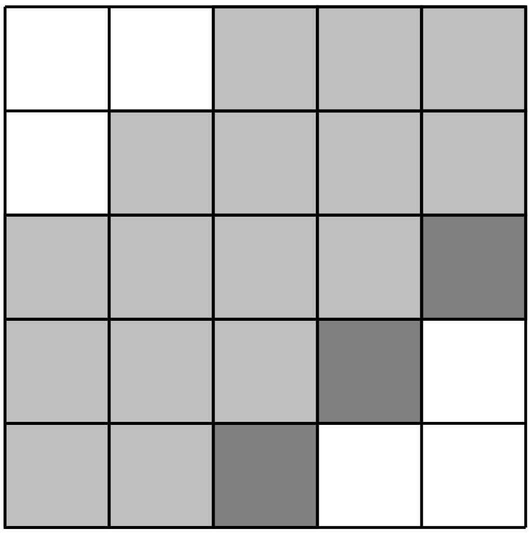


Fig. 4.4 Dependencies of the wavefront when solving the knapsack problem, once the wavefront is more than halfway through the scoring grid

maintained. This is shown in Fig. 4.3, where the dark grey cells are the wave front, and the lighter grey cells are the dependencies.

Once the wave front has moved passed the centre point, only then can old iterations begin to be transferred off the GPU. Figure 4.4 shows how the dependency structure appears when the wavefront has moved beyond the halfway point of the scoring grid. It can be observed that some cells are maintained that are not needed, for example the entire top row of the scoring grid. This is due to the fact some of the cells of the iterations they belong to are required by the wavefront, and therefore the entire iteration must be maintained.

Due to the higher number of previous iteration dependencies, memory complexity for data which must be stored on the GPU is $O\left(\lceil\frac{W\cdot n}{2}\rceil\right)$ where W is the capacity of the knapsack and n is the size of the item set. Test data is stored on the GPU in two vectors, a vector containing the weight of all items in the input data set, and a second vector containing the profit of these items. In the case of the bounded knapsack problem, a third vector is stored on the GPU dictating how many times each item can be selected. All of these vectors reside in constant memory of the GPU. As the memory usage of the algorithm is dependent on the capacity as well as the size of the item set, a limit to the size of input problem cannot be defined in terms of item set size alone. However we seek to provide an example of a problem which will use all of the memory of the test GPU. A problem instance for the 0/1 knapsack problem which contained 30,000 items and had a knapsack capacity of 400,000 would be a rough limit for the problem size which could be executed on our GPU. This is based on the assumption all data structures are storing standard 4 byte integers.

4.2 Available General Optimisations

There are few global optimisations that can be made at the high level of overall model design, as most are problem dependent and should be input by the user.

As already mentioned, during the rotation kernel, all threads are synchronised before read and write operations take place to global memory. Due to this, read and write requests to global memory are guaranteed to take place simultaneously. By ensuring the threads are in synchronisation, and the memory request is completely linear, it allows the memory transactions to be coalesced, meaning multiple memory operations can take place in a single cycle. Also, when rotating the memory, the function which translates the indexes of requested cells from the scoring grid, to actual indexes in memory is not used – the rotation operation is hard-coded and the user has no control over this.

The size of the blocks spawned are already optimised through the memory management model, based on the target block size defined by the user. Enough blocks are launched to contain the wave-front, and the wave front is padded as required to ensure it is a multiple of the block size. Therefore, there are no optimisations available with respect to the size of the blocks.

A key point that the user must be aware of when implementing the dynamic programming case, is to limit operations that require data from previous iterations, as these are stored in global memory of the GPU. A small point, that can cause considerable improvements on the run-time is once the user has requested values they should be stored locally within a thread local variable. For example, if a value is needed in multiple calculations, it should be stored locally within the thread once it has been accessed initially, rather than being loaded from global memory multiple times. Across the execution of the algorithm, this has a large effect on the runtime as all user accesses to global memory are required to use our wrapper to determine the correct indices.

Finally, we take steps to ensure the heavily used memory index look-up function runs quickly. In this function, when calculating the requested indices we ensure that the GPU math library is used at all points, enable hardware based math operations to take place providing the highest level of performance. Again, any small gain provided here will have large impacts across the algorithm, as this function is used so extensively.

4.3 Specialised Problems

During the course of the development and implementation of this algorithm, some problems were found to be unsuitable for implementation using the proposed parallel model. Here we discuss these limitations in an effort to find a formal definition as to which problems are, and which problems are not applicable for use.

4.3.1 The Travelling Salesman Problem

The travelling salesman problem required significant adaption to be solved by our model, moving away from our principle of inputting new problems through a file format, and requiring hard coded adaptations to the model. However, we sought to continue to use the same principle of using the wavefront method, so the approach we adopted uses the same algorithmic approach.

Based on the dynamic programming definition, once the scoring grid has been created, to fill each cell a new subgrid is required to be solved. Then to fill each cell of this subgrid,

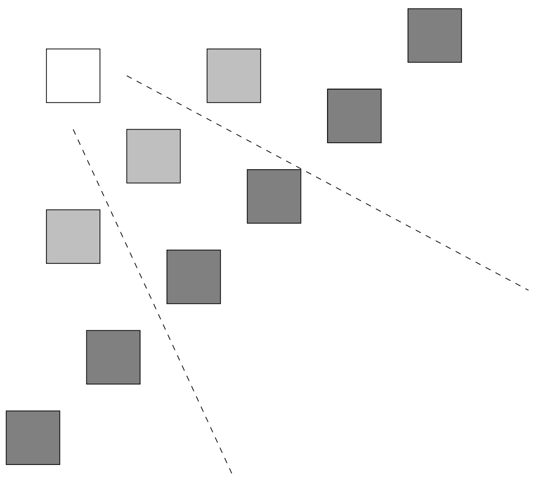


Fig. 4.5 Multiple subgrids are required to solve the travelling salesman problem

further subgrids are required and thus recursion continues. This is shown in Fig. 4.5, where the master grid requires 3 subgrids, which in turn require more. Solving the problem through this method is an example of a bottom up approach, where the smallest subgrids must be solved first, allowing progressively larger grids to be solved, until finally the single largest scoring grid contains the overall answer. This diagram also demonstrates that this can still be represented in a diagonal based form, similar to our wavefront approach.

To solve this problem, instead of using the GPU threads to fill cells in parallel, we use the threads to solve entire scoring grids in parallel. Starting with the smallest scoring grids, this creates a very wide front, all of which can be solved in parallel (the dark grey cells in the figure). As these scoring grids complete, they pass values up to the next level of grids (the ones in light grey in the figure), and these are then all solved in parallel. In this manner the travelling salesman problem can be solved, with the wave front moving from very wide iterations to progressively smaller. However, this increases the issue of iterations becoming not wide enough to utilise all the resources of the GPU, as the the size of the wavefront reduces in size. Also as iterations take longer to complete, it is possible that a larger proportion of the run-time of the algorithm is spent calculating wavefronts that are smaller than the GPU has the resources for.

We considered splitting scoring grids such that each subgrid is solved my multiple threads rather than a single thread, as the wavefront becomes progressively smaller, yet this would have caused considerable implementation complexities. Also, this continues to move our model further away from the original proposed design methodology, therefore we elect to simply have one thread solving one scoring grid. There is a benefit associated with this approach that due to the reduced thread count for the final iterations, there is more memory available to each thread. As these iterations are also computing larger subgrids, the additional memory is beneficial to the algorithm in the final stages of execution.

Our memory management model can still be used with this problem, again requiring some adaptation. As the subgrids are computed, the memory can be pushed off the device to the host as it is no longer required. However, this data must then be transferred back to the GPU as the algorithm moves up a level of scoring grids. This means that the user has no control over whether or not data is stored on the host. As calculation of scoring grids is completed, and the results successfully transferred back to the GPU for the next level of calculation, these can be removed from memory on both the host and the GPU. This reduces overall memory complexity and allows larger problems to be solved.

The travelling salesman problem required adaptions to the model to allow it to be solved, but we believe the approach we have adopted is closely related to the proposed model, and we claim it is still solved through the same method.

4.3.2 All Pairs, Shortest Path APSP Problem

The all pairs, shortest path problem is also a challenging problem to solve. Due to its dynamic programming definition each cell must check whether there is a shorter route between two vertices by going via every other vertex, causing dependencies between every cell in the scoring grid. This leads to the situation in which no old diagonals of the wave front can be transferred back to the host memory as they are all be required by future iterations. Therefore the problem size is heavily constrained by the amount of memory on the GPU, only allowing adjacency matrices of size $40,000^2$ or smaller to be solved.

The only methodology we could identify to resolve this would be to split the scoring grid down to smaller blocks, solving these in isolation, and communicating results afterwards. Whilst this is similar to the the approach we chose for the travelling salesman problem, in the case of the APSP problem, this is a larger digression from the approach of a wavefront, and moving closer to existing block decomposition methods readily available in the literature.

Therefore at this point, we choose not to implement the APSP using our model, as the adaptations required would offer little research contribution, and define this problem as unsuitable for solving via the base model. However during development and testing we did implement a reference APSP implementation [51] for investigation. This was used to support the work of the research group which has been submitted, but not yet published, in IEEE Transactions on Evolutionary Computation.

4.3.3 Defining Unsuitable and Inefficient Problems

Based on the implementation issues we encountered, we observe that some problems are well suited for implementation using the proposed model, others less so or requiring adaptation, and some which are unsuitable for implementation at all.

The dependency structure of the method used to solve the problem is key in defining whether a problem can be solved by our model or not, and how efficiently it can be solved. Broadly, problems fall into four categories of suitability for solving through our model:

- *Ideal* - These are problems that require a small number of previous iterations to be maintained as dependencies of the current iteration being solved. This has the advantages of allowing large problem instances to be solved due to low memory requirements on the GPU, as well as ensuring the GPU has a high ratio of computation operations compared to memory transaction operations. Example of these problems include the longest common subsequence, knapsack, edit distance, and Manhattan tourist problems.

- *Inefficient* - Problems which require a large amount of previous dependencies to be maintained limit the size of instances that can be solved, as more iterations are required to be stored on the GPU - but they can still be solved successfully. An example of a problem in this category is the subset sum problem, where the current iteration has dependencies to all previous iterations prior to the current one. In this case, the model will still solve the problem successfully, but no previous iterations can be moved off the GPU, meaning no memory management will occur, considerably limiting the size of solvable problem instances.

- *Requires Adaptation* - If a problem has dependencies ahead of the wave front, it means they are unsuitable for implementation without an adaptation to support this. To allow cells ahead of the wave front to be available to the current iteration, the current iteration should be kept in memory as a previous iteration until the future cells it requires are reached by the wave front; at this point these can be filled, and the iteration rotated off the GPU. Examples of problems that fall into the category are the all pairs shortest path, and chain matrix multiplication problems.

- *Unsuitable / Requires Extensive Adaptation* - Finally, some problems are unsuitable to be implemented through the model without extensive changes being made - these are generally problems which require recursive child scoring grids to be created, such as the travelling salesman problem, or have otherwise specialised problem specific solving methodologies. Whilst we have demonstrated solving the TSP problem in this thesis using the same basic paradigm as our model, it required extensive hard coded adaptions to enable this.

Summary

In this chapter we have described the mapping of the test problems onto the GPU through the use of our parallel model. We have demonstrated through the use of the input files and changing the dynamic programming statement, different problems can been solved. Also discussed were adaptations to the model required when solving problems with more complex dynamic programming algorithms. Finally, problems which are unsuitable for implementation were considered, and the reasons for this are presented.

Chapter 5

Testing Methodology

Overview

This chapter describes the hardware environment used for carrying out the performance testing of the proposed model, and the testing methodology adopted when analysing the program. Covered is the hardware specification of the machines used during testing, as well as the software that was used during both the compilation and execution of the program. We introduce the metrics that will be recorded during testing execution, and describe how these can be used to evaluate the model performance, as well as detailing how the test data is generated for each introduced problem. Finally, we provide some small scale benchmarks demonstrating the performance of the underlying hardware, giving theoretical best case performance that the model could achieve.

5.1 Testing Environment and Hardware

In this section the hardware and software environment used across all testing is described.

The majority of the testing took place on a desktop computer running the Arch Linux operating system. The code was compiled using the GNU Compiler Collection (GCC) 4.8, and GPU code was compiled using version 6.5 of the CUDA framework. The machine has an Intel 3930K CPU providing 6 cores clocked at 3.2GHz, and 16GB of DDR3 memory. It also contained 2 NVIDIA Titan GPUs which each provide 2688 CUDA cores and are clocked at 837 MHz. Note however, no multiple GPU tests were carried out as the model has not been designed to support this, instead the multiple GPUs just enabled simultaneous testing in the smaller test cases.

A secondary desktop with a lower specification GPU was used in order to demonstrate how the code scales from one hardware environment to another. Code on this machine was compiled using GCC 4.6, and the GPU code was compiled using version 6 of the CUDA framework. In terms of hardware, this desktop has in Intel i7 4790 providing 4 cores clocked at 3.6Ghz, and 16GB of DDR3 memory. It also contains a NVIDIA GTX 960 providing 1024 CUDA cores clocked at 1127 MHz. A full description of the role of the second system during testing will be described shortly.

5.2 Methodology

Now we will detail the testing methodology we use to validate the performances of the proposed model.

5.2.1 CUDA Metrics

The CUDA software environment ships with a built in profiler *nvprof* [73] that allows an end user to retrieve accurate metrics relating to the CUDA code that is running on the GPU.

As NVIDIA CUDA is a closed source, proprietary programming environment, this the only profiler that allows a developer to record performance information during execution of a CUDA kernel. This is advantageous however, as this profiler is produced by NVIDIA, it allows for the use of hardware counters producing high quality profiling results. Listed here are the metrics we will be recording during all test runs of the proposed model.

1. *Global load efficiency* - The ratio of the requested memory load operations from global memory compared to the amount that took place. Therefore when considering the results of this metric, a higher figure is better. It should be noted that it is possible for this figure to increase past 100% if multiple threads within a single warp are requesting memory from the same address.

2. *Global store efficiency* - Identical in operation to the previous global load metric, but instead of considering memory load operations, it records memory store operations.

3. *Warp execution efficiency* -The ratio of the active threads present in a warp compared to maximum amount of threads that can be active in a warp based on the multi-processor on the GPU hardware. Again, with this metric, higher is better.

4. *Branch Efficiency* - The ratio of the branches of the program that are spawned during execution which are non-divergent, compared to the total number of branches that are created. For this metric, a higher value is more desirable.

5. *Achieved Occupancy* - Ratio of the number of active warps per processor cycle, compared to the possible amount of warps that can be active per processor cycle. For this metric, a higher value is better.

6. *Instructions executed per cycle* - The number of instructions executed for each clock cycle of the processor.

Now we consider our rationale for selecting each of these metrics. The results of metrics 1 and 2 are important as they relate to the efficiency of memory operations from the global store. As discussed, the global store is the slowest of all of the CUDA memory locations, and if used incorrectly can prove to be a bottle-neck on the run-time of a GPU kernel. Therefore whilst we seek to reduce the amount of memory transactions to the global store, it is impossible to avoid them entirely, so great care should be taken to make sure they run efficiently. Metrics 1 and 2 are a direct reflection on this, and should they come back with low results, it will demonstrate that the run-time of the kernel is being dominated by waiting for memory transactions to take place.

Considering the warp execution efficiency (metric 3) is important, as it gives an indication to how effectively the available computational resources of the GPU are being used. As there is a finite limit on the amount of warps that can be active on a multi-processor at a time, it is important that warps that are running contain as many threads as possible. Should this metric prove to be low, it shows that there is divergence in the code, and portions of the GPU are simply idling rather than executing.

The branch efficiency (metric 4) of the program is important, and is linked to the warp efficiency. Remember that all threads within a warp must follow the same code path, therefore for maximum efficiency the divergence of the algorithm must be kept to a minimum. This metric is a direct reflection on the amount of divergence in the code, and as with the warp execution efficiency if this value is low it may mean that some of the resources on the GPU are idling. However, it should be noted that some divergence can be present without affecting the overall efficiency, if the divergent branches are using a number of threads equal to the size of a warp – in this case both code paths are still executing at maximum efficiency.

At a higher level than these is the metric that considers the achieved occupancy, metric 5; instead of counting how many individual threads are running in a warp, it considers the amount of warps that are actually running. Many people consider this metric to be one of the most important [42], as it gives a quick snapshot overview of how efficiently an algorithm is likely to run on the GPU. Maximising the warps that are running at a given time is key to hiding the latency of the kernel on the GPU. As such NVIDIA have even developed a occupancy calculator [72] which gives a crude estimation of this metric before runtime. The occupancy, and metric 5, is heavily influenced by the parameters such as the block size, and the amount of threads and memory available on the underlying hardware. This is discussed further when we consider the testing environment for each individual problem in Section 5.2.4. A low value in this metric could mean a variety of things, such as bad algorithm design, structuring the size of the blocks badly or failing to configure the launch parameters appropriately for the hardware. Therefore, investigation using more fine grained metrics such as branch efficiency and warp execution efficiency is required.

Finally, the instructions executed per clock cycle (metric 6) gives an overall indication to how efficiently the resources on the GPU are being used. This metric shows how much work the processor is actually performing for each cycle of the processor clock. This means that should there be a lot of time the cores are idling, this metric will reflect this and will then allow for more detailed investigation to be carried out. We believe that using this subset of metrics allows us to gain an accurate insight as to how effectively the model executes in terms of memory performance, processor performance, and adherence to an effective CUDA programming paradigm considering factors such as divergence.

5.2.2 Empirical Metrics

We also use some more common metrics to gauge other aspect of the models performance, listed here:

1. *Wall time* - A record of the total execution time each experimentation run requires. We record this metric in terms of seconds.

2. *Memory used (host)* - A record of the total peak amount of system (non GPU) memory used during an experimentation run. We record this metric in terms of megabytes.
3. *Memory used (GPU)* - A record of the total peak amount of memory used in the GPU global memory store during an experimentation run. We record this metric in terms of megabytes.

Recording the wall time is one of the most common metrics when measuring the performance of any newly proposed algorithm or model. However we believe it is of limited usefulness, due to the fact it is a highly hardware dependent measure, varying from system to system. It is even possible for the run time of an algorithm to change between systems using the same hardware, due to differences in the software environment, and configuration. In an ideal situation, other algorithms from the literature would be re-implemented on the same system to produce comparative runtimes, but this is simply in-feasible and outside of the scope of this work. Therefore we provide run times to demonstrate to the reader how long it takes to solve a given problem size, and more importantly the cut off when the problem size becomes to large to be solved in an *acceptable* time frame, but direct comparisons to other studies won't be drawn. In instances where other studies are suitably recent, reference run times from these may be provided for illustrative purposes.

As with recording run time, recording memory usage is a metric that is applicable when measuring the performance of any algorithm. With the overall scoring grid being stored in memory on the host, based on the assumption there is more available than on the GPU, it is important to record how large this value can rise to. Should this prove to be the limiting factor, this will also provide an absolute value for the maximum size a problem instance can be, before this model is no longer applicable.

Similarly, we record the amount of memory used on the GPU to ensure that the technique of moving data back to the host is effective, and the GPU has enough resources available to run the model. The rate at which this metric grows can also give an indication as to at which point instance of problems become to big to be calculated. This metric has to be recorded through specialised NVIDIA tools however to ensure an accurate value is recorded.

5.2.3 Global Testing Parameters

All runs are replicated 20 times, as although during testing we ensure that all machines experimentation is taking place on are otherwise idle, there may still be some background noise from other running processes. Also, should there be any unexpected or unusual results, as the runs have been replicated it can be ascertained this was not a quirk of one specific run, and rather a quirk of the algorithm as a whole.

The CUDA block size used during execution also needs to be fixed between runs. Rather than using a different block size between problems, we elected to use a uniform block size across all runs as this means there is minimal change to the underlying code of the model between tests, and this gives a fairer insight to the applicability of the model between problems. As detailed, finding the block size is a factor that is heavily dependent on the amount of memory the algorithm being executed needs, as well as the amount of resources that are available on the specific GPU. For our testing we fixed the block size at 512 – this is a multiple of the warp size of the GPU which is 32, meaning maximum efficiency at this block size can be achieved. Also, setting the block size to 512 means each block has a maximum of 4096 bytes of memory available to it, and each thread has 32 registers available, which we expect to be a comfortable amount for our test problem implementation.

5.2.4 Test Data

In this section we describe how test data is generated for the performance testing, and which parameters are adjusted across different runs.

The Longest Common Subsequence Problem & Edit Distance

For the longest common subsequence problem the variable parameters are the length of the input strings, and what percentage of the strings match to form the longest common subsequence. All test data for this problem was generated by ourselves, by first fixing the length of the two strings, then defining the length and content of the shared longest common subsequence. The longest common subsequence was then placed randomly in both strings, maintaining ordering, then the rest of the strings are filled using random characters from the defined alphabet. For all of our test data we used the alphabet A, C, T and G as this is the alphabet of DNA bases commonly used in computational biology, and one of the most common uses of this problem.

For testing we varied the length of the strings between 0.5 and 3 million, in steps of 0.5 million, and varied the amount of commonality between the strings from 0 to 100% in steps of 25%. As well as running tests with strings of the same length, tests were also ran when there was a length difference between the strings – when one string was twice as long as the other, and when one string was 4 times longer than the other (i.e. one was 25% the length of the other). Increasing the length of the string allows the efficiency of the algorithm to be observed as the size of the input data changes, and allows for investigation into whether the parallel scaling of the algorithm changes as different input sizes are used. Changing the length difference between the two strings means that the dimensions of the scoring grid changes, and may have an impact on metrics such as warp efficiency, and warp divergence. Finally changing the length of the longest common subsequence within the strings should have no effect on the run time, or any other metrics, but this should be validated through testing.

In the case of the edit distance problem, for ease of implementation we use the same generated test data as that which is used for the longest common subesequnce problem. This gives a good range of string size, as a well as range of edit distances. Due to the similarity between the problems, fluctuating the test data of the edit distance problem, should have a similar effect on the recorded metrics as with changing the data for the longest common subsequence problem.

The Knapsack Problem

In the case of the knapsack problem, the adjustable parameters are the set of items which can be selected from, the number of times each item that can be selected, and the capacity of the knapsack. As with the longest common subsequence problem, we generate our own test data. Although there is standardised test data available for the knapsack problem, it is dated, and targeting CPU implementations so proves not sizeable enough to warrant execution on the GPU. Item sets are generated at random with 80% of the items having weight in the range of 5-20% of the total capacity of the knapsack, 10% in the range of 0-10% of the capacity and the final 10% in the range 20-30%. All items had profit values assigned to them at random between 0 and 100.

For testing we varied the capacity of the knapsack from 10,000 to 100,000 in steps of 10,000. The number of items in the associated item set was varied from from 0.8% to 1.6% of the capacity of the knapsack, in steps of 0.2%. Changing the capacity of the knapsack will affect the size of the scoring grid in one dimension, and changing the number of items will change the size of the scoring grid in the other dimension. This will have an effect on a whole range of metrics; firstly as it grows in either dimension there will be associated increase in run time and memory usage, whilst there is also likely to be changes in warp execution efficiency and branch efficiency as the size of the problem passes optimal block size boundaries.

We use the same test data for both the bounded and 0/1 version of the problem. For the bounded implementation, we assume that a vector x of equal length to the item set, stores a list of the amount of times each item can be selected. This vector is filled at random with positive integers.

The Manhattan Tourist Problem

For the Manhattan tourist problem the adjustable parameters are the size of the grid, as well as the weights defined upon the edges. As with the previous problems, we generate our own bespoke test cases for this problem due to a lack of availability of online test cases, seeking to adjust both of the above parameters simultaneously. Throughout testing the grid is maintained as square, with the length of both dimensions increasing equally as they are altered.

Generating the test data was the simple case of altering the size of the grid from 5,000 to 30,000 in steps of 5,000 and generating values for the edges as random numbers within the range $0-100$.

Travelling Salesman Problem

The travelling salesman problem is arguably the most challenging problem we consider, and also requires an adapted implementation to make the model applicable, as covered in Sect 4.3.1. When solving TSP instances through the classical methods of heuristics, test data can have an effect on the run time, not simply in terms of size but also in terms of the placement of the cities, meaning there is a very strong concept of *best case* and *worst case data*. However in our implementation, as we seek to solve instances exactly, the only factor that should impact the run-time is the number of cities being considered in the problem.

Rather than generating test data for the TSP, we instead use standardised data that is available in the online repository TSPLIB [81], as here we can find suitably large instances which warrant GPU implementation. The test data we use from the library is: A280, ATT532, PR1002, NRW1379, U2152m, FNL 4461, where the number within the test data name denotes how many cities the instance contains, as this gives a good range of sizes for testing. Note that we stop testing at the relatively small instance of 4461 cities, as the memory constraints of this algorithm are so incredibly high, even when our memory management features are employed.

Changing the data in this manner will have an effect predominately on the run time, and also on metrics concerned with memory usage. It will be interesting to identify if the load and store efficiency changes considerably between runs as the test data changes, as these metrics are more likely to be linked to the implementation of the TSP problem, rather than affected directly by the test data. Also the warp execution efficiency and branch efficiency is likely to change marginally based on the test data, as with the other problems, but we do not expect considerable changes between runs solving different problem instances.

5.3 Comparative Data

When developing a model for deployment on a GPU, finding a method to allow comparative testing between our model and the literature is challenging as it is beyond the scope of the work to implement a plethora of different competing GPU and CPU algorithms for each individual test problem. However, to illustrate the effectiveness of our approach we must consider its performance in the context of similar works.

Therefore the approach we have elected is in some instances to implement competing algorithms, and in others to use results directly from the literature, whilst ensuring the shortcomings of using other authors results are discussed here. Firstly we consider sequential CPU implementations. It is expected that a GPU implementation that is taking advantage of thousands of cores, should be orders of magnitude faster than a sequential, single core CPU implementation. Therefore, in the case of single core CPU algorithms, using results from the literature is acceptable as they are only used to illustrate the fact our GPU implementation is a clear improvement over a CPU based method, and we do not use the results for direct comparisons between the run times. The same is also true of parallel CPU based implementations, where it is expected the GPU should comfortably outperform the run time of such algorithms. Again, in these cases, we draw results straight from the literature which use parallel CPU based implementations, to hypothesise a rough figure for the amount of CPU cores that would be required to deliver a similar level of performance to our GPU method.

When we are comparing against competing GPU algorithms by using the results directly from other publications, care must be taken not to draw direct comparisons between comparative runtimes, or other metrics, due to the difference in execution environment. Therefore, in the following chapter presenting the testing results, we discuss our findings in the context of other GPU algorithms. This provides a rough framing for where our algorithm fits within the wider range of available algorithms, but we avoid drawing comparisons based on values such as the percentage improved run time offered by our model compared to other implementations, or similar.

Now we will identify the data which will be using from the literature for comparison.

5.3.1 Sequential CPU

As the longest common subesequence problem is a classic problem, CPU implementations from the literature can be very dated, sometimes going back 30 years which obviously are inappropriate for even rough comparison against. Due to this we implement our own version of a basic dynamic programming approach to the longest common subsequence problem. To complement this with more advanced algorithms, a relatively modern survey paper [11] re-implements a whole range of longest common subsequence algorithms, including some diagonal based methods [96, 67, 64], which we also use for comparison.

As with the longest common suebsequence problem, simple, sequential CPU based algorithms for solving the Edit Distance Problem, and the Manhattan Distance Problem are not present in modern literature. Therefore as creating basic dynamic programming algorithms for these is relatively simple, we also create our own simplistic CPU based implementations of these. It should be noted that in all our CPU implementations the size of the test data had to be considerably limited due to the memory requirements of these algorithms, and a memory management model similar to the GPU for the CPU was not implemented.

For the knapsack problem there is a recent survey paper [61] which focuses on more modern approaches to solving this problem. Therefore as well as using a basic dynamic programming approach of our own implementation, we also briefly compare against the results of the dynamic programming method of Pisinger [77], as implemented by Martello [61].

The travelling salesman problem is unique, as it is almost exclusively solved through inexact methods for larger problem sizes. Therefore for this implementation we implement the same dynamic programming on the CPU as the one which is implemented on the GPU, as we believe this will provide good illustration of the benefits GPU can offer, and how our model makes exact methods feasible again.

5.3.2 Parallel CPU

To compare against a CPU parallel implementation of the longest common subsequence problem we use the work of Krusche and Tiskin [54], which is a modern implementation of the earlier bit-parallel algorithm by Crochemore [21] which iterates through the grid in a wave front. and describes considerable optimisation's compared to older algorithms. This means there are multiple levels of parallelism present in this algorithm – there is bit-parallelism where cells of the scoring grid are represented by bits, as well as the higher level block synchronous parallelism when large chunks of the scoring grid are calculated in parallel. Bit-parallelism is the concept that by storing multiple cells in a single data type, for example an integer or a long, only one operation needs to be carried out to affect all of them. Therefore we expect this algorithm to be high performing, and a representative example of a parallel CPU algorithm.

Due to the similarity of the implementation of both the edit distance problem, and the Manhattan tourist problem, we elect to use the results from the Krushe and Tiskin paper as an indicator of the performance of these problems if they were to be implemented on the CPU in parallel.

For the parallel CPU implementation of the Knapsack problem we consider the survey paper of Rashid et al. [80] which is a modern paper concerned with analysing the performance of multiple parallel Knapsack implementations, with a focus on reanalysing the performance of classic parallel algorithms [35, 34].

5.3.3 Parallel GPU

For GPU parallel algorithms, we can use the works identified in the literature review in Sect. 2.4.3. For comparison of the longest common subsequence problem we compare our implementation against the work of Kloetzli et al. [53], whose implementation paradigm is similar to ours. We also compare our implementation with Yang et al. [99], who demonstrate the restructuring of data dependencies in the scoring grid. As with the CPU implementation, we can also use this as an indicator of the performance of the edit distance problem and the Manhattan distance problem.

We compare our knapsack problem implementation against the GPU parallel work of Boyer et al. [14], who demonstrates a row parallel method of solving on the GPU.

For both the parallel CPU and parallel GPU implementation, we do not compare the yravelling salesman problem to printed results, as we cannot compare our exact solution to inexact solutions from the literature, due to the fact this would require the consideration of factors such as solution quality.

5.4 Reference Benchmarks

In this section, reference benchmarks for the introduced metrics are provided to give the reader an indication of baseline performance in a perfect environment, allowing conclusions on how comparatively efficient the proposed model is.

To create these benchmarks we use a specifically crafted test problem that is designed to offer perfect conditions for the model to operate in, with test data size lining up perfectly to block boundaries, and being accessed in a perfectly sequential manner. This test problem is very similar to the longest common subsequence problem, however only one input piece of test data is used. As the wave front iterates through the grid, the only dependency the cell has is the one directly to the left of it (in the previous iteration), as this will be in the same column within the data structure of previous diagonals being stored on the GPU. The cell is filled based on the following very simple case:

$$REF_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ REF_{i-1,j} + 1 & o.w \end{cases} \tag{5.1}$$

A cell of the input test data is also read with the matching i value to simulate a read operation taking place. Input data is simply a vector of initialised with zeros and the scoring grid is assumed to be square, with i being the same size as the length of the input data. Block size was set at 512 to mirror our real testing parameters.

The test problem was then deployed on the two described desktops (one with a NVIDIA Titan, the other with a NVIDIA GTX 960), and the metrics were recorded across five replicate runs.

We can see from the Fig. 5.1, that the metrics for the load and store efficiency metrics are extremely high, nearly universally higher than 90%, demonstrating that in perfect conditions the memory management of the model is highly effective. An interesting point of this however is the fact the metrics do not pass 100%, as this is possible for the load and store values. If

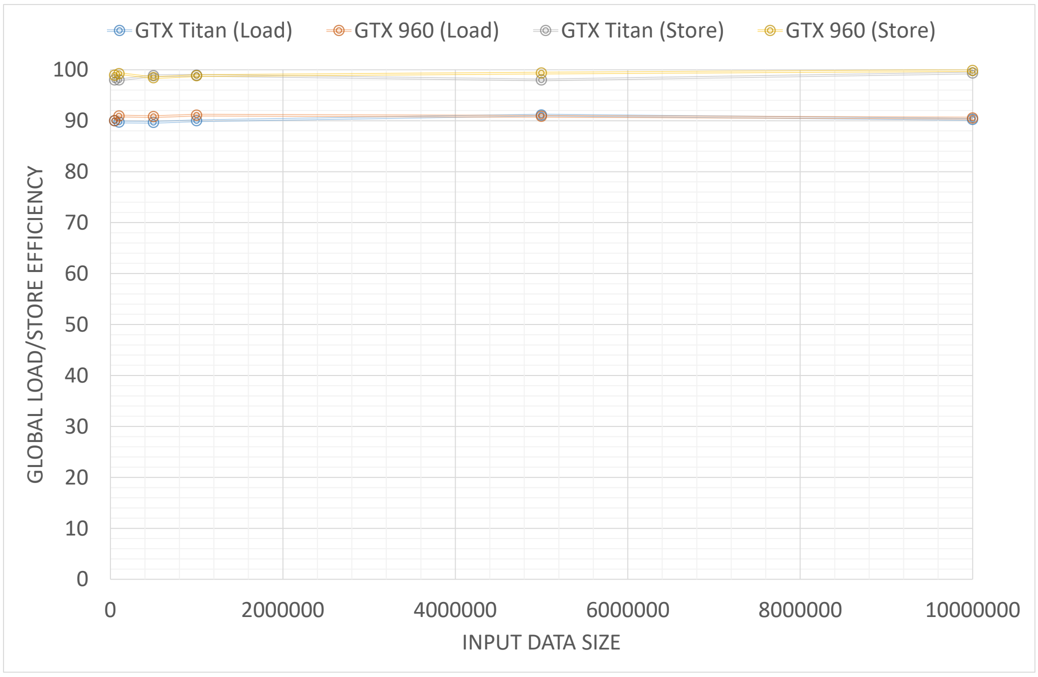


Fig. 5.1 Efficiency of the global load and store operations whilst running the reference implementation, as the size of the scoring grid increases

they were to pass 100% it would show that multiple threads were accessing the same address simultaneously, which would be assumed to be the case as loading the test data take place the same *i* value. We hypothesise this is due to the wrapper that translates the memory addresses is obviously causing work on the thread, which will in turn induce a delay before the memory transaction takes place, and therefore all threads are not perfectly synchronised before the memory operation. Without all threads being synchronised, a coalesced memory request cannot take place.

In Fig. 5.2 the other recorded CUDA metrics are presented. This shows that the warp efficiency averages at roughly 80% and the branch efficiency at roughly 90%. Again these are very strong initial values for the efficiency of the model. Initially it was expected that the warp efficiency and branch efficiency would have been more intrinsically linked. We assume the warp efficiency is lower than the branch efficiency due to the fact that warps can stall within already divergent branches, reducing the efficiency of the executing warps further.

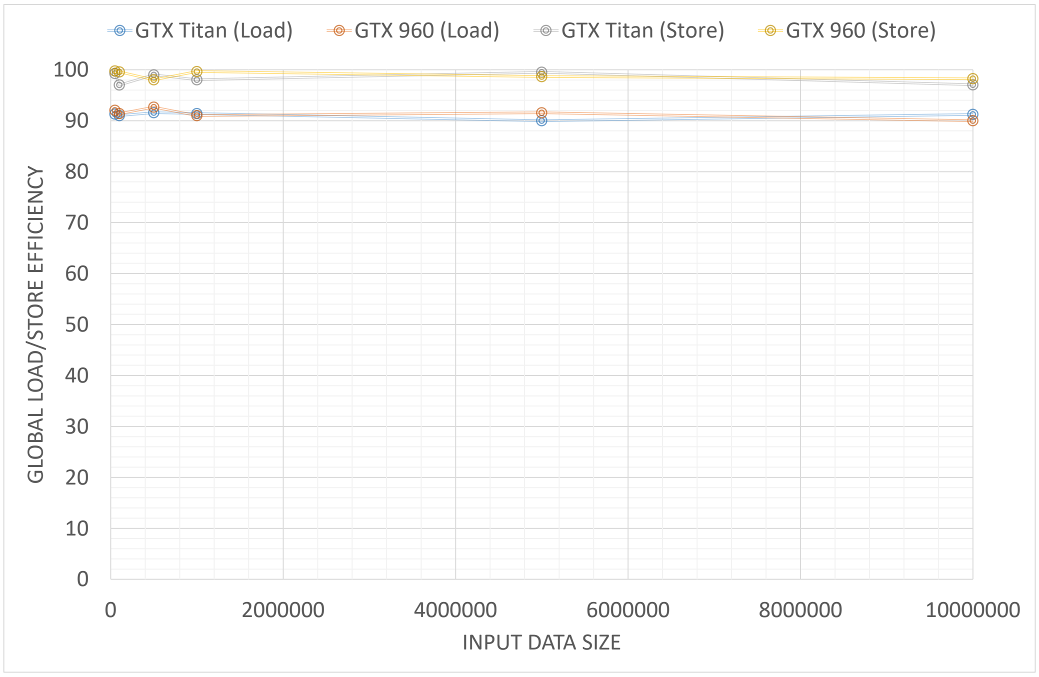


Fig. 5.2 Recorded CUDA metrics whilst running the reference implementation, as the size of the scoring grid increases

The recorded value for occupancy is very high reaching nearly the perfect value of 100%, which shows all the resources of the GPU are in use and also demonstrates that our choice of block size is effective. Occupancy alone cannot be used as an indicator though, as whilst it shows all resources are being used, they may not being used effectively.

A fact observable from both of the graphs thus far is the GTX 960 seems to perform marginally more efficiently than the GTX Titan. This is likely due to micro-architecture improvements on the GTX 960, as whilst in terms of CUDA cores it trails the Titan by a considerable distance, it is a more modern example of GPU hardware. The exact reason for the small discrepancy however cannot be isolated. The similarity in performance between the two pieces of hardware demonstrates strong scaling and portability of the model, and its performance isn't bound to a specific piece of hardware.

Summary

This chapter has presented details of the two systems that will be used for the performance analysis of the model, a higher powered, but older Titan based machine and a more modern GTX960 based system. The methodology used when testing the model through the defined test problems was discussed with the selected metrics introduced, and justification provided as to why we are using a heavily metric based analysis method. Also presented are methodologies for generating test data for each of the problems, as well as relevant results from the literature which we will use to demonstrate the effectiveness of our model. Finally we present some theoretical benchmarks as to peak performance the model can offer in perfect conditions. The next chapter details the results of testing the model following the methodology laid out here.

Chapter 6

Performance Testing

Overview

This chapter describes the performance testing of the proposed model using the test problems, evaluated through the introduced metrics, using the discussed testing methodology. The results of the testing are discussed, and the performance results are justified. Comparisons are also drawn against results from the literature to illustrate the benefits of the proposed model.

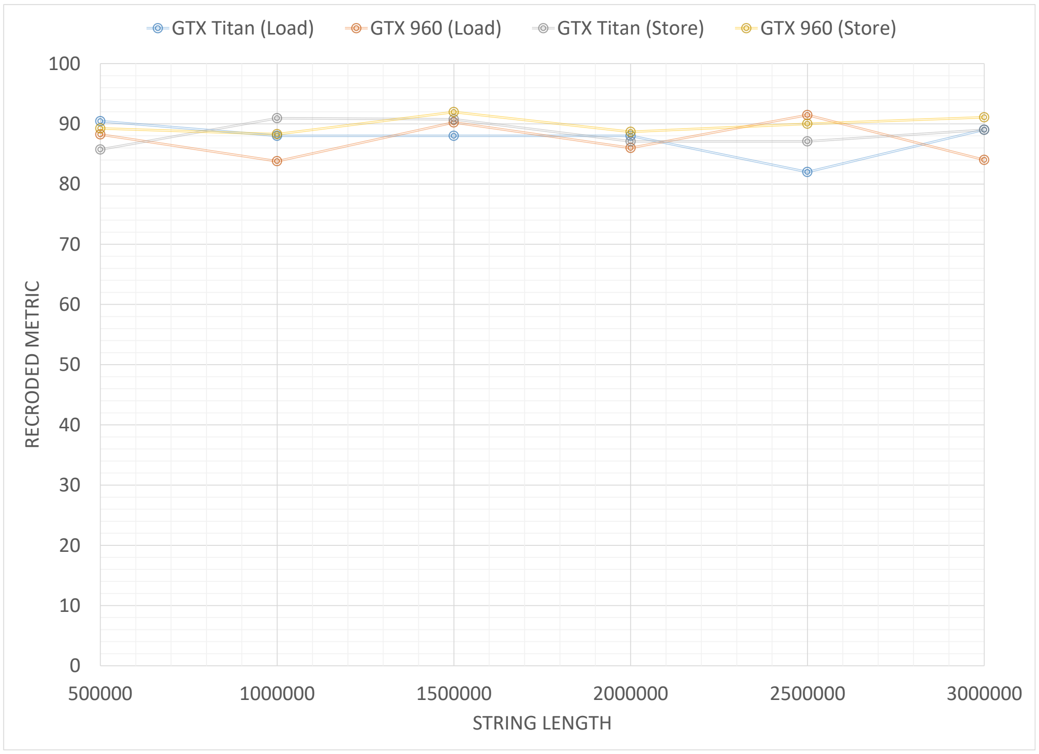


Fig. 6.1 Recorded global load and store efficiency whilst solving the longest common subsequence problem, on two strings of equal, increasing length, when the match is 50% of each string

6.1 Performance on the Test Problems

The section presents results of the models' performance when running on the introduced test problems.

6.1.1 The Longest Common Sub Sequence Problem

We begin by testing the longest common subsequence problem. As defined in the testing methodology, we solve the problem for strings of increasing length, different proportions of common subsequence, and varying length differences between the input strings. Graphs and more detailed analysis is provided for testing when there is a 50% common match between the two strings, and the two input strings are of the same length.

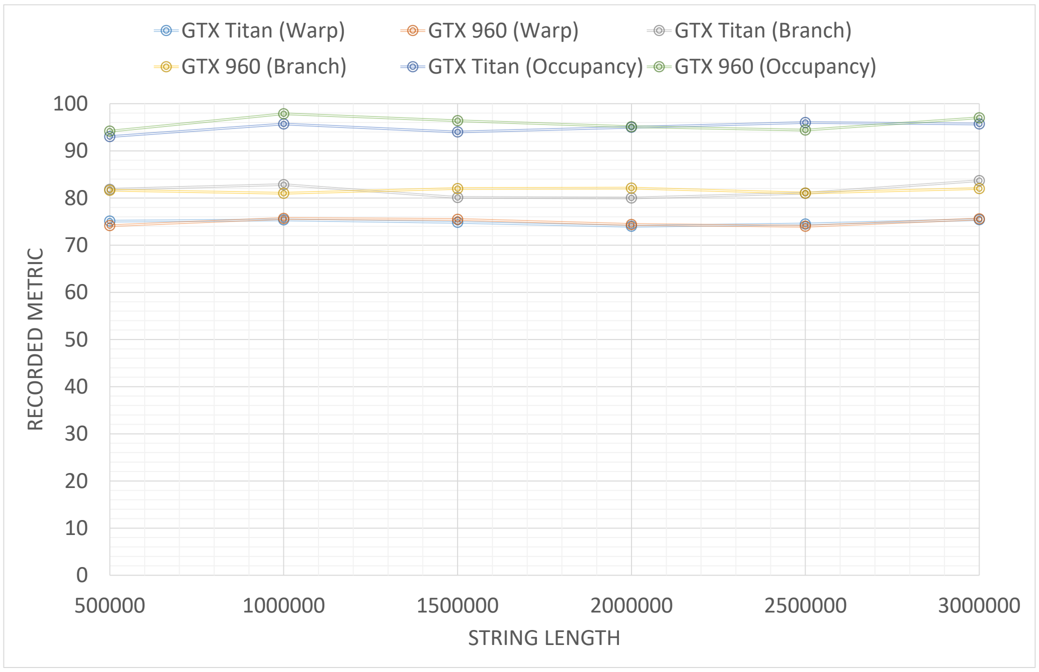
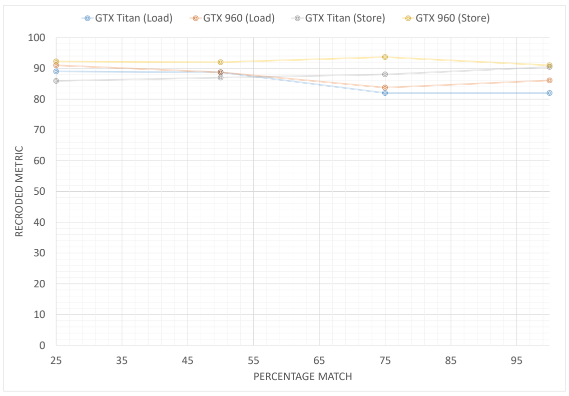


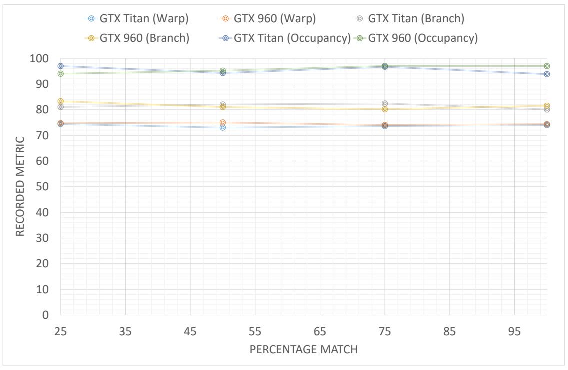
Fig. 6.2 Recorded CUDA metrics whilst solving the longest common subsequence problem, on two strings of equal, increasing length, when the match is 50% of each string

Beginning with the global load and store efficiency metrics, Fig. 6.1 shows the recorded values. These results have a mean of 90% which is a strong result. A point to note however is the results appear to be slightly dropping, as the size of the string grows, when running on the older hardware. The recorded results are also marginally lower than the reference figures for the efficiency of storing in global memory – this could either be during the rotation kernel, although this is unlikely as threads are in perfect synchronisation at this point, or during the main execution kernel. Therefore we assume this is linked to threads taking a different amount of time to perform work, and writing the result back at different times, potentially causing other threads to wait.

The other recorded CUDA metrics, with the exception of instructions per clock (IPC), are presented in Fig. 6.2. The graph shows the warp efficiency at roughly 73% for both devices, the branch efficiency averaging at 82% and the occupancy at 95%. As with the previous test these values are high, and very encouraging, but somewhat short of the reference values. This is to be expected as the reference values were constructed to demonstrate a theoretically



(a) Efficiency metrics for load and store operations as the percentage match between the two strings is altered



(b) Other CUDA metrics as the percentage match between the two strings is altered

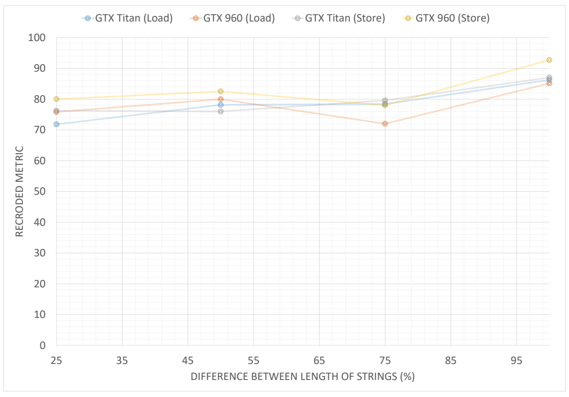
Fig. 6.3 Metrics recorded when solving the longest common subsequence problem with a fixed string length of 3,000,000 and the longest common subsequence between the strings is varied between 25% and 100%

perfect implementation of the model, and is reflective of real world usage. The pattern of falling short of the reference values is prevalent throughout testing, as detailed in this section. A further promising result from this graph is the results for both the GTX Titan and the more modern GTX960 are comparable, which gives an initial indication that the model adapts well to a varying number of CUDA cores and to different CUDA API levels.

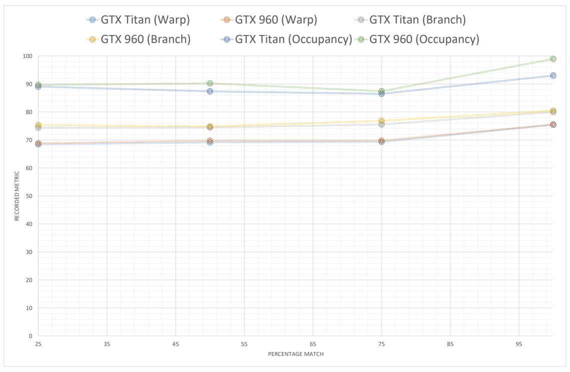
Varying Match Percentage

We now consider the effect that altering the percentage of common subsequence shared between the strings has on the performance of the model. These tests are expected to provide near identical results to the previous tests, as the match percentage should only affect the values being stored within the scoring grid. The length of the string is fixed in these tests at 3,000,000.

Figure 6.3a shows the global load and store efficiency metrics, and Fig. 6.3b shows the other recorded CUDA. As expected these results match the trends of the previous runs almost exactly. This demonstrates the performance of the model is not dependent on the input data.



(a) Efficiency metrics for load and store operations as length difference between the two strings is altered



(b) Other CUDA metrics recorded as the length difference between the two strings is altered

Fig. 6.4 Metrics recorded when solving the longest common subsequence problem with a fixed string length of 3,000,000 for the first string, and the second string length is altered between 25% and 100% of this

Fluctuating String Length

Next we consider the affect that altering the size of the length differential between the two strings has on the models performance. This test will cause the scoring grid to become non-square, and therefore the model will be required to perform more rigorous bounds checking, and the code will likely have to diverge more often. Due to this we expect the branch efficiency and the warp execution efficiency to suffer. The impact on this metrics may not be in direct proportion to the string length differential, the efficiency depending instead on how many warp boundaries are crossed on each iteration.

Figure 6.4a details the load and store metrics for two strings of differing lengths, and Fig. 6.4b shows the other recorded CUDA metrics. The length of the first string is fixed at 3,000,000 and the length of the second string is altered between 25-100% of this. It can be seen that there is an efficiency impact on the model when the two strings are not of identical length causing the scoring grid to be non square. The observed efficiency cost is minor with the biggest impact being on the load and store memory transactions reducing the recorded

Table 6.1 Average instructions per clock recorded during the performance testing of the longest common subsequence problem

String A Length	String B Length	Match Percentage	IPC
500000	500000	50	0.54
1000000	1000000	50	0.55
1500000	1500000	50	0.55
2000000	2000000	50	0.54
2500000	2500000	50	0.55
3000000	3000000	50	0.55
3000000	3000000	25	0.53
3000000	3000000	50	0.55
3000000	3000000	75	0.55
3000000	750000	50	0.47
3000000	1500000	50	0.44
3000000	2250000	50	0.45

metrics by between 15-20% in the worst case. Warp and branch efficiency are also impacted but to a lesser extent reducing by between 5-7% in the worst case.

These are promising results as warp and branch efficiency, and occupancy, are metrics that can begin to fall away dramatically as the code diverges, as clearly stated in the NVIDIA CUDA documentation.

Instructions per Clock

Finally we consider the instructions per clock recorded whilst running the longest common subsequence problem in the different testing scenarios.

The average values for the instructions per clock metric are presented in Table 6.1. The results shown are highly consistent if somewhat low. We believe this is due to the highly memory bound nature of our model, where the performance of the algorithm is dependent on the speed of the memory transactions rather than CPU operations. The algorithm can be required to pause whilst memory transactions are taking place, causing the IPC value to fall. We are not discouraged by these results however, as whilst the IPC value is somewhat low,

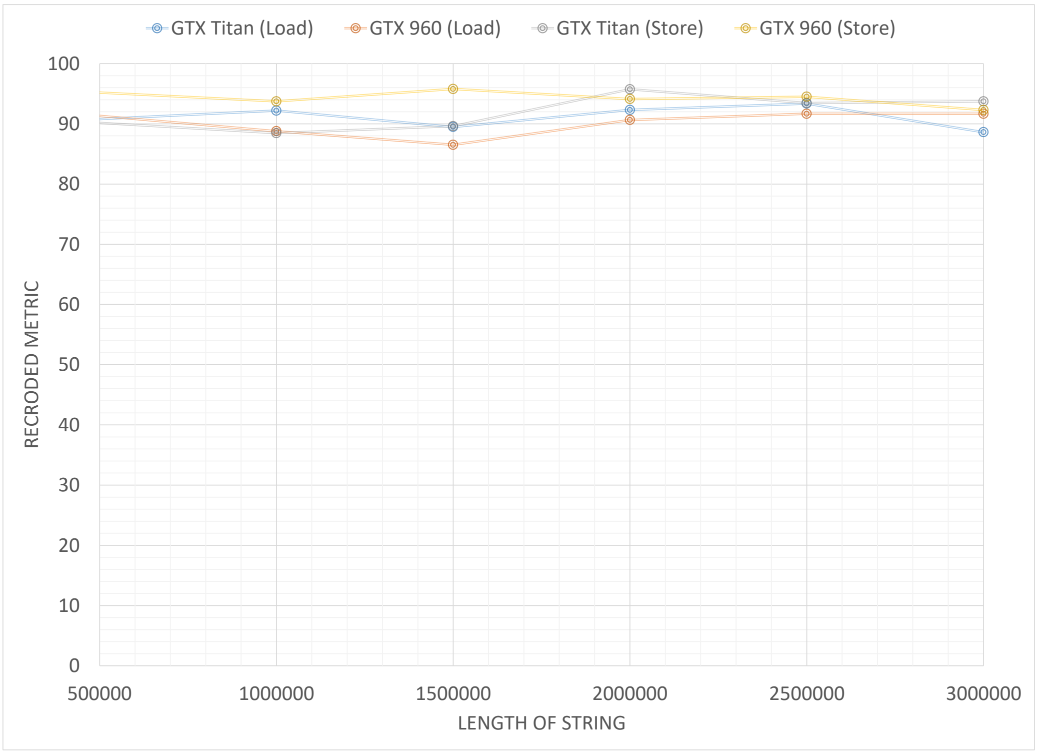


Fig. 6.5 Recorded global load and store efficiency whilst solving the edit distance problem, on two strings of equal, increasing length, when the match is 50% of each string

the previously recorded metrics are high, indicating that whilst memory speed is the limiting factor, the memory based portions of the model are performing efficiently.

6.1.2 Edit Distance Problem

We next consider the edit distance problem. Due to the similarity between this and the previous problem we use the same testing methodology as outlined in Sec. 5.2.4. We also therefore expect similar results to the longest common subsequence problem, with the algorithms only differing with the edit distance problem performing more work to fill each cell. Whilst the problems are similar, these tests serve to demonstrate the model running on a different problem that has been defined through the input files.

As with the longest common subsequence problem we begin testing with strings of equal length, increasing in size, and a common subsequence length of 50%. Figure 6.5 shows the

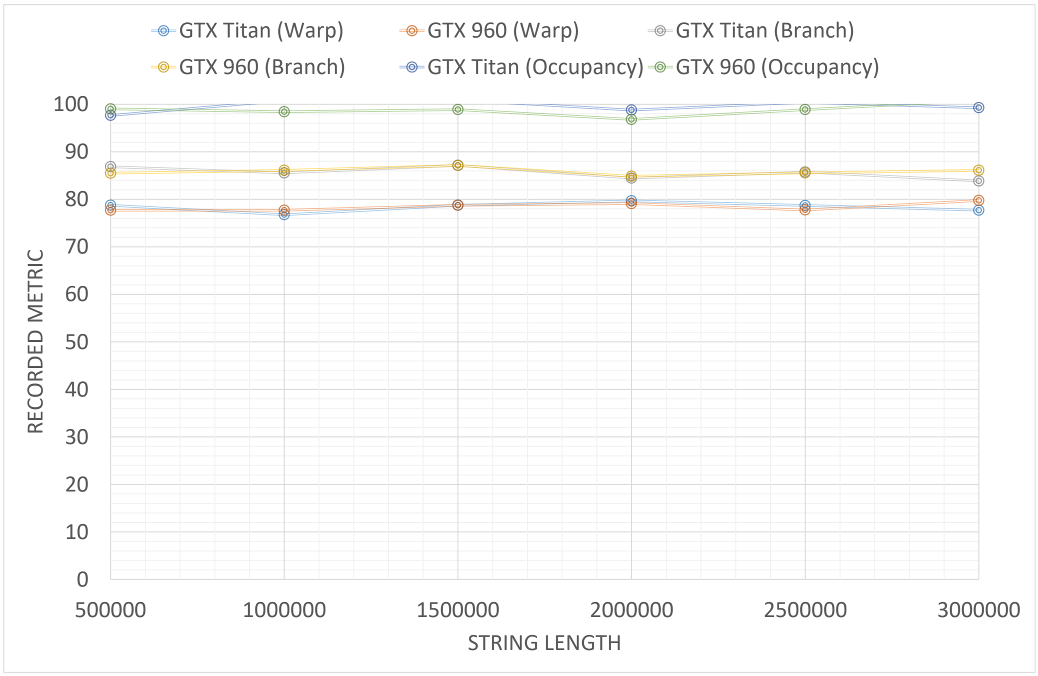
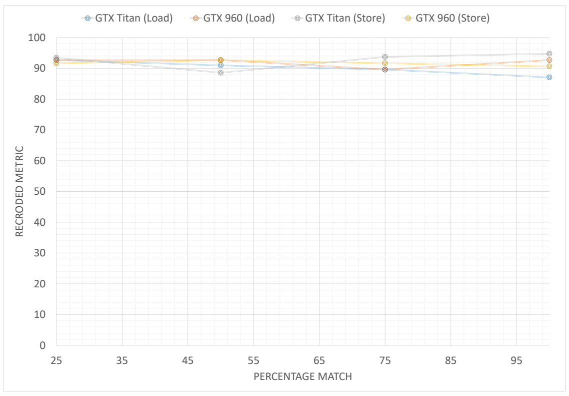


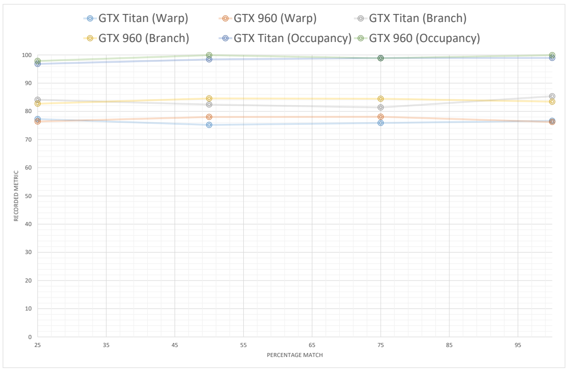
Fig. 6.6 Recorded CUDA metrics whilst solving the edit distance problem, on two strings of equal, increasing length, when the match is 50% of each string

recorded efficiency metrics for the memory transactions recorded during these tests. The first result of note is the metrics tend to be between 2-4% improved over those of the longest common subsequence problem. Whilst this is a minor performance gain, this is an accurate result due to the replicate runs. We believe this is further indication that the model is heavily memory bound, as when there is even a small amount of additional computation to fill each cell, the observed efficiency of the model rises. Secondly, the results show the same pattern of being relatively constant, with the performance of the model unaffected by the length of the string. As these values all average near 90% or higher, these are extremely promising results for our model.

Figure 6.6 details the other recorded CUDA metrics for the edit distance problem. These also show improved results over the longest common subsequence problem, but of greater magnitude than the memory metrics, reaching nearly a 5% improvement in some cases. This is a more interesting result as divergence, and occupancy based metrics were expected to be



(a) Memory efficiency metrics for load and store operations as the edit difference between the two strings is altered



(b) Other CUDA metrics as the edit distance between the two strings is altered

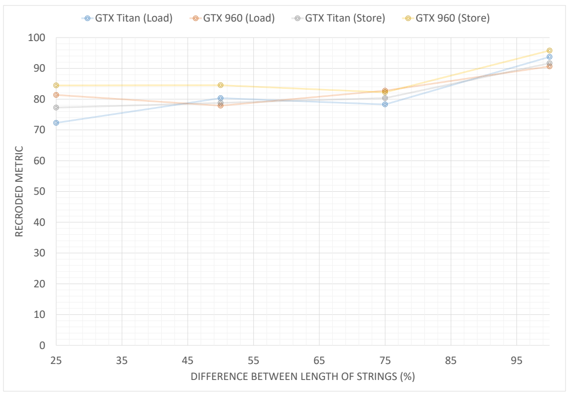
Fig. 6.7 Metrics recorded when solving the edit distance problem with a fixed string length of 3,000,000, and the amount of commonality between the strings is varied between 25% and 100%

unaffected by the change from one problem to the next, especially given that the strings are still of the same length. The slightly improved performance on these metrics is difficult to account for. Note also that the occupancy results are especially high, approaching 100% in some cases. These results show strong performance of the model, and demonstrates it can move effectively from one problem definition to the next.

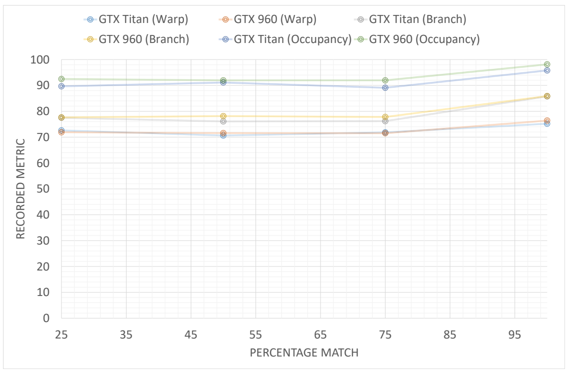
Varying Edit Distance

We now vary the match percentage between the strings, which will in turn change the final resulting edit distance. As before, this is expected to have no effect on the recorded metrics, as the final answer output should not impact the performance of the algorithm. Figure 6.7a shows the memory store and load metrics, and Fig. 6.7b the other recorded CUDA metrics for these tests. As expected the amount matching between the strings has no affect on the observed metrics which remain constant and inline with the previously observed results.

At this point we consider the fluctuations that have been present in most recorded results thus far, where results have followed trends but not given straight lines when plotted. These



(a) Memory efficiency metrics for load and store operations as the length difference between the two strings is altered



(b) Other CUDA metrics recorded as the length difference between the two strings is altered

Fig. 6.8 Metrics recorded when sovling the edit distance problem with a fixed string length of 3,000,000 for the first string, and a second string length altered between 25% and 100% of this

are not due to noise, as all tests were run multiple times to eliminate noise, therefore these fluctuations are in fact present throughout testing. We hypothesise these are due to the problem instance having an affect on the run times. For example, in the presented problems so far, the code is required to take different paths based on whether it finds a matching character or not, therefore the structure of the generated test data is could be causing the observed fluctuations. As the same test data is used across each replicate run, the fluctuation is not removed through repeated testing. We assume if multiple runs with different test data took place with fixed string lengths, and fixed commonality between the two strings, the fluctuations could be smoothed out.

Varying Length

Next, we consider tests where the input strings are not identical lengths, causing the scoring grid to become non square. As before, we fix the first string with to length of 3,000,000 and the second string length is altered between 25-100% of this. CUDA metrics recorded for these tests are presented in Fig. 6.8, with the memory efficiency metrics shown in Fig. 6.8a

Table 6.2 Average instructions per clock recorded during the performance testing of the edit distance problem

String A Length	String B Length	Match Percentage	IPC
500000	500000	50	0.58
1000000	1000000	50	0.58
1500000	1500000	50	0.58
2000000	2000000	50	0.59
2500000	2500000	50	0.58
3000000	3000000	50	0.58
3000000	3000000	25	0.57
3000000	3000000	50	0.58
3000000	3000000	75	0.58
3000000	750000	50	0.51
3000000	1500000	50	0.52
3000000	2250000	50	0.54

and the other CUDA metrics in 6.8b. Continuing the pattern of testing for the edit distance problem, the results follow very similar trends to the longest common subsequence problem where by using strings of different lengths has upto a 5% impact on the observed efficiency metrics. Overall performance is marginally higher for this problem than than in the case of the longest common subsequence problem.

Instructions per Clock

Finally, we consider the instructions per clock recorded during the performance testing of the edit distance problem. These results are interesting in the context of the slightly improved performance of this problem, and may offer some clarification as to why.

Table 6.2 shows the recorded values across testing. As before, the IPC values are very consistent showing little change between tests, except a small reduction when testing was taking place on strings of different lengths. A noticeable result of the testing however is the IPC value is universally higher than it was during the LCS testing, although the algorithms are very similar. This shows that a small change in the problem definition, and how the cell is filled, can have an impact the efficiency of the algorithm.

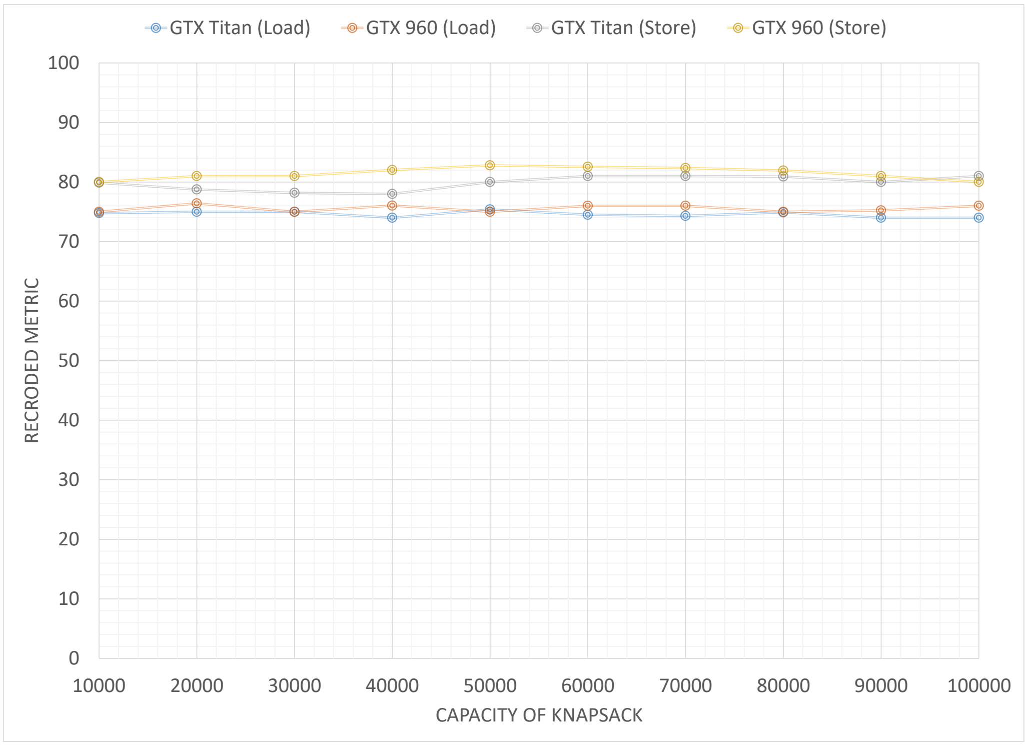


Fig. 6.9 Recorded global load and store efficiency whilst solving the 0/1 knapsack problem, as the capacity of the knapsack and associated item set increases

6.1.3 The Knapsack Problem

Next we consider a more distinct problem, the Knapsack problem, which serves to further demonstrate the applicability of the model to different problems. This problem allows us to gauge the performance of the model when significantly more previous iterations are required to be maintained locally on the GPU than has been required in previous testing.

For the first tests we record the same sets of metrics as used in previous testing, whilst running the 0/1 knapsack problem with differing capacities of knapsack. The size of the item set is fixed at 1% of the capacity in all tests, and the values of the items within the set are generated as defined in Sect. 5.2.4. These tests are expected to show considerably different results to the ones observed thus far, as the knapsack problem has a more challenging memory access pattern, accessing memory in a more randomised manner, and the scoring grid will be non-square in all instances.

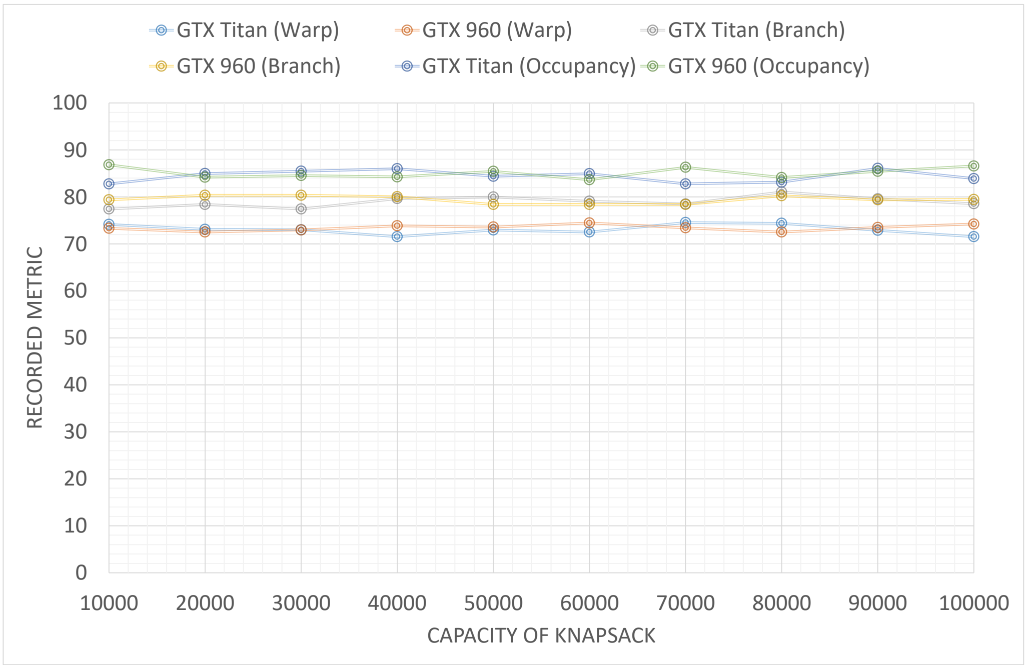


Fig. 6.10 Other recorded CUDA metrics whilst solving the 0/1 knapsack problem, as the capacity of the knapsack and associated item set increases

Recorded metrics for the global load and store efficiency are presented in Fig. 6.9. The first observation drawn from the graph is the fact the results are upto 15% lower than the longest common subsequence problem in the worst case. This is likely due to the fact that less memory can be accessed sequentially when loading profit values from across the scoring grid. However, this does not account for the reason that the store efficiency metric is also lower, which is expected to be in line with previous results as the wave front traverses through the grid in the same diagonal manner.

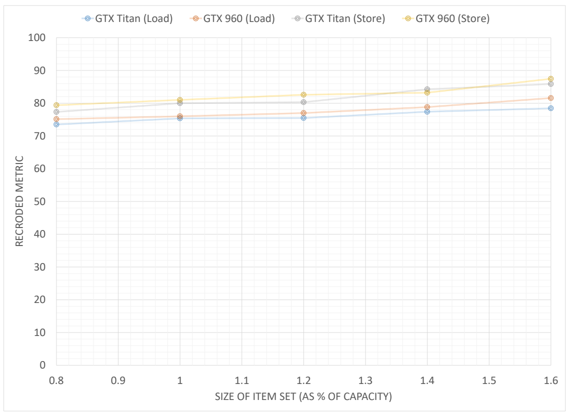
The second point observed from the graph is that there is a smaller differential between the load and store metrics, and there is also a smaller difference between the performance offered by the different hardware configurations. The reduced difference between the hardware likely indicates some memory transactions are reducing the performance of the model, and using a more modern CUDA device cannot alleviate this issue. The smaller difference between the load and store is likely due to waiting on slow memory transactions – for example memory cannot be stored back into the wave front, until first previous memory has been loaded, causing the store operation to stall. Overall, it is apparent that the efficiency of the load and store operation is impacted in the case that memory accesses are required across a high number of different previous iterations.

In Fig. 6.10 the other recorded CUDA metrics are presented. Compared to previous testing, the occupancy metric has reduced by upto 8% in some instances. This indicates that the GPU is not executing as many warps as it could be, and thus the GPU is not being kept busy. However, in contrast to this, the recorded metrics for the branch and warp divergence are very similar to those observed in previous testing, if marginally reduced. These results reinforce the theory that the memory access pattern is having an impact on the overall efficiency of the model. As occupancy has been reduced, but the branch and warp metrics have stayed similar, the negative impact on the occupancy therefore must have been caused by waiting for memory transactions.

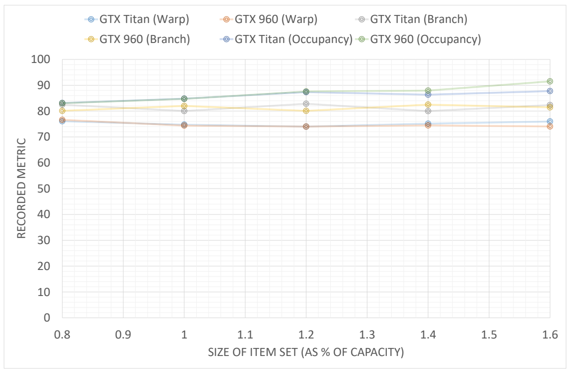
Varying Item Set Size

Next we consider changing the size of the item set that is used during testing, whilst keeping the capacity constant at 50,000. Changing the amount of items within the set, will have the affect of altering the scoring grid in a different dimension to changing the capacity. Therefore we would expect similar results to previous knapsack problem testing.

Results of these tests are presented in Fig. 6.11, with the load and store metrics in Fig. 6.11a and the other CUDA metrics in 6.11b. These results present an interesting pattern not observed thus far, of an upwards trend in the memory efficiency metrics, and the occupancy metrics as the size of the item set increases. Also note, the same trend is not observed in either the branch or warp efficiency metrics. As the item set grows, the scoring grid becomes closer to being square, therefore this is likely the root cause for the small performance gains. Due to this, we assume that as the grid become more square, less bounds checking is required and cells from previous iterations can be accessed more efficiently improving the overall



(a) Memory efficiency metrics for load and store operations as the size of the knapsack item set is altered



(b) Other CUDA metrics recorded as the size of the knapsack item set is altered

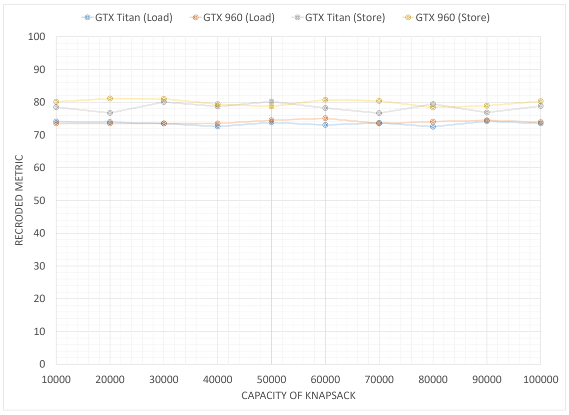
Fig. 6.11 Metrics recorded when sovling the 0/1 knapsack problem with a fixed capacity of 50,000, and the size of the item set is altered between runs

efficiency of the model. This would also provide explanation as to why the longest common sub-sequence problem appears to perform more efficiently, as across testing of that problem, the scoring grid was often perfectly square.

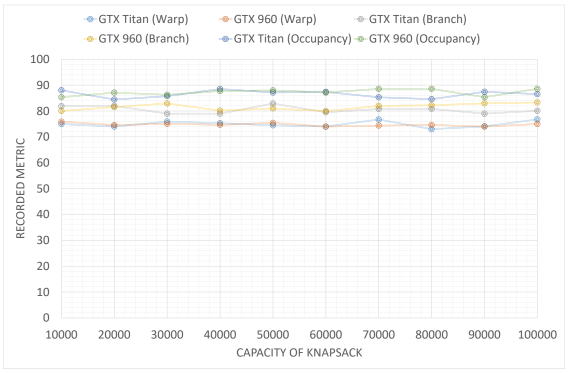
Bounded Knapsack Variant

Next, we run the bounded variant of the knapsack problem, where each item is permitted to be picked x_i times, where x is a vector of integers equal in length to the size of the item set. We assume in this case, that results should be very similar to those observed in the previous knapsack testing. Changing the problem variant further demonstrates that the model can be adapted for a range of problems and uses. As with the initial 0/1 knapsack problem testing, the size of the item set is fixed at 1% of the capacity.

Load and store metrics for these tests are provided in Fig. 6.12a and results for other CUDA metrics are showing in Fig. 6.12b. As expected, these results show very similar values to the previously run experiments, although marginally better across all tests. The reason for this very small improvement is difficult to isolate, although it is most likely due to the simpler problem definition of the bounded version.



(a) Efficiency metrics for load and store operations as the capacity of the knapsack instance is altered



(b) Other CUDA metrics recorded as the capacity of the knapsack instance is altered

Fig. 6.12 Metrics recorded when running the bounded knapsack problem with varying capacities, and a fixed item set size of 1% of the capacity

Instructions per Clock

Finally we consider the instructions per clock values recorded whilst testing the knapsack problem. The recorded average IPC values are given in Table 6.3.

Observed in the table are some of the lowest values for the instructions per clock recorded during test. For the initial tests focusing on the 0/1 variant of the problem, the IPC value drops below 0.5 for the first time. This shows that when running the knapsack problem, the model spends time stalled waiting for memory transactions to complete. Therefore, whilst the model is applicable to multiple different problems and variants, the inherent memory access model the problem requires has an effect on the observed performance. It should be noted however that the user is free to restructure memory requests to behave in an optimal manner, the end user has control over how much data is stored in the GPU, and what cells are loaded for each iteration of the wave front, and where results are written back. Therefore if the user can design a more complex and efficient memory access pattern, they are free to do so through the API design of our model. This implementation of the knapsack problem is a very simplistic example, and can be improved upon.

Table 6.3 Average instructions per clock values recorded during the performance testing of the 0/1 knapsack problem

Capacity	Item Set Size	IPC
10000	100	0.49
20000	200	0.48
30000	300	0.49
40000	400	0.49
50000	500	0.5
60000	600	0.49
70000	700	0.5
80000	800	0.5
90000	900	0.51
1000000	1000	0.5
50000	400	0.49
50000	500	0.5
50000	600	0.5
50000	700	0.52
50000	800	0.51
Bounded		
10000	100	0.52
20000	200	0.52
30000	300	0.52
40000	400	0.53
50000	500	0.52
60000	600	0.51
70000	700	0.52
80000	800	0.52
90000	900	0.51
1000000	1000	0.51

6.1.4 The Manhattan Tourist Problem

Next, we consider another distinct problem, the Manhattan tourist problem. Whilst the application of the Manhattan tourist problem is quite different to the previous problems tested, the core of the algorithm similar to that of the longest common subsequence problem. As is the case with the LCS problem, the data is represented as two pieces of input data. In this case though, the input data consists of the vertical and horizontal weights for each column and row of the grid respectively.

We therefore expect similar results to those of the longest common subsequence problem, however it is still beneficial to show the model adapting to a further problem.

Figure 6.13 shows the recorded global load and store efficiency for the Manhattan tourist problem. There are two interesting features of the graph – firstly, these are the highest recorded metrics observed since the reference benchmarks were carried out, and these results

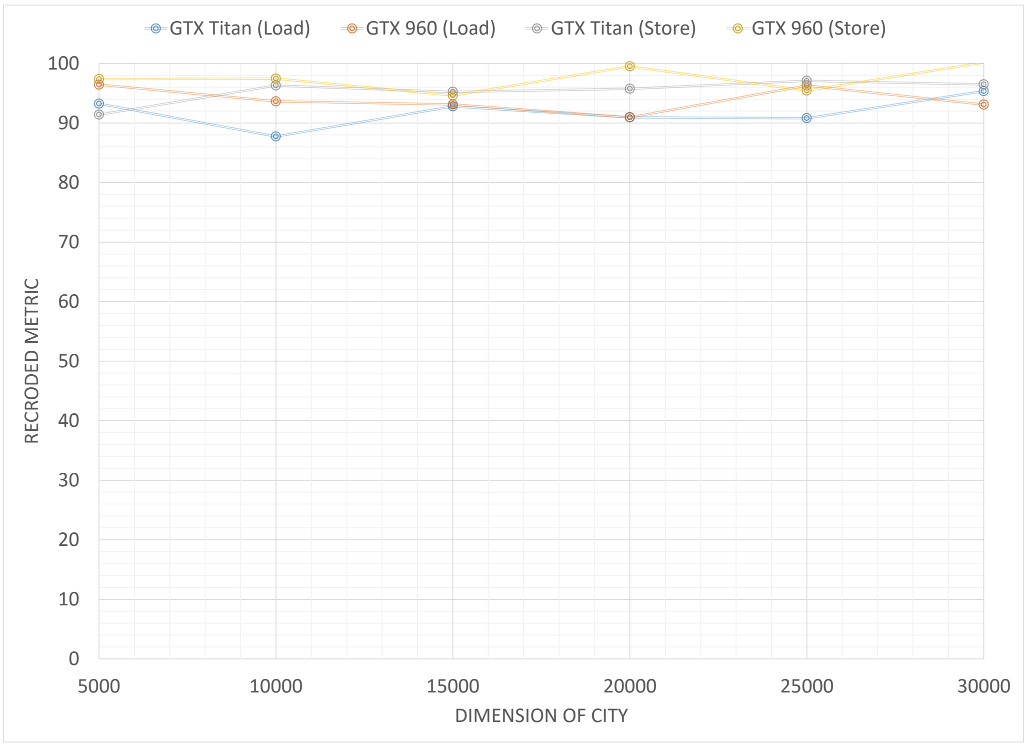


Fig. 6.13 Recorded global load and store efficiency whilst solving the Manhattan tourist problem, as the dimension of the city grid increases

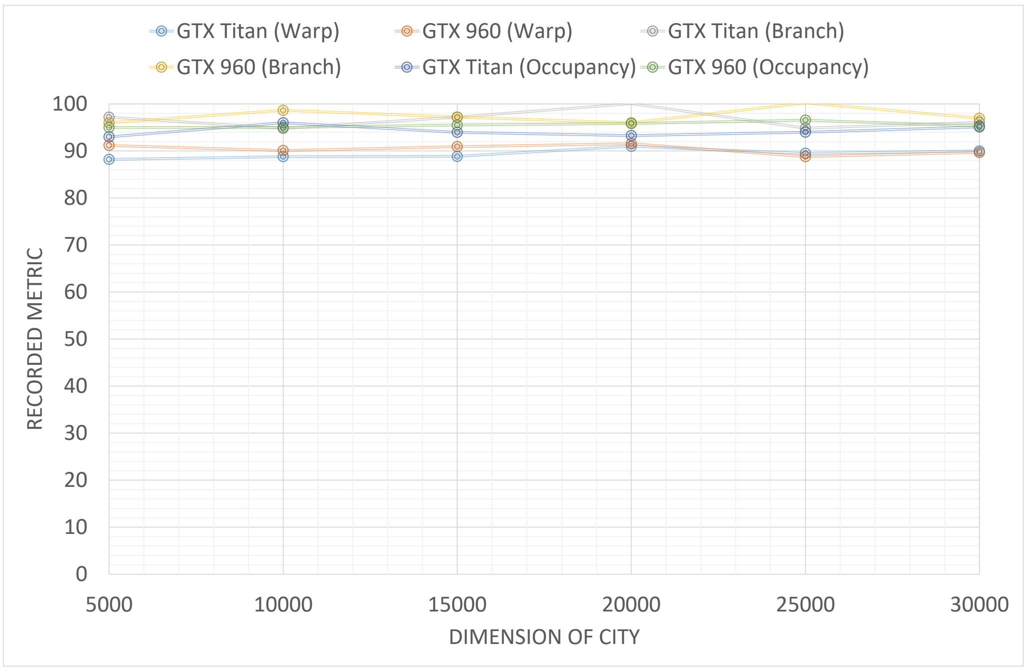


Fig. 6.14 Other recorded CUDA metrics whilst sovling the Manhattan tourist problem, as the dimension of the city grid increases

also seem to show the largest variance between steps fluctuating as high as 10% between steps of increasing size.

Firstly we consider the reason for the strong performance the model demonstrates running this problem, and what the cause of this is likely to be. From the problem definition it can be seen that this problem has a very simple memory access pattern, only requiring data from the previous iteration, and a single piece of data from each of the two pieces of input data. This is a similar memory structure to that of the longest common subsequence problem only differing in the number of previous iterations required by one, therefore is not enough alone to account for the improved performance. The other difference between the problem definitions is the presence of an `if` clause in the core definition of the longest common subseqence problem, and there is not one in the Manhattan tourist problem. This indicates the lack of branching in this problem is providing a performance improvement, which is aided by the reduced number of memory transactions.

The fluctuations are very challenging to explain. As with previous tests, we can run multiple tests at each point using different problem instances to smooth the trend, but using the same test data repeatedly at each point leads to the non-smooth graph presented. We must therefore assume that this problem is input data dependent, however it is hard to identify where as there is minimal branching within the problem definition which could have this affect. Due to this the only explanation remaining is some input data sets are structured more favourably for the CUDA execution engine due to the values they contain.

The second graph which details the other CUDA metrics of occupancy and warp and branch efficiency, presented in Fig. 6.14 demonstrates a similar picture. All of the metrics are extremely high, approaching the values attained during the reference benchmarks. The lowest recorded metrics are the warp efficiency at 90%, and the branch efficiency and occupancy average at 95% or above. This reinforces our earlier assumption that the improved performance of the algorithm is predominately due to the lack of branching present in the

Table 6.4 Average instructions per clock values recorded during the performance testing of the Manhattan tourist problem

City Dimension	IPC
5000	0.66
10000	0.65
15000	0.66
20000	0.66
25000	0.65
30000	0.66

core problem definition. Another point to observe from this graph is the fluctuations which were present in the memory efficiency graph are far less prominent now. This potentially indicates that the reason for the fluctuations in the memory graph is a hardware specific result of loading and storing the required values, as the other metrics are far more stable.

In the previous test problem, the knapsack problem, we observed that performance was reduced due to the memory access pattern, now we observe that performance is considerably improved based on the structure of the problem. This indicates that the performance of the model is sensitive to the structure of the underlying problem.

Instructions Per Clock

Finally, we consider the instructions per clock metric for the Manhattan tourist problem which are given in Table 6.4. These values remain constant, at the highest values observed so far throughout the testing.

Based on the last two problems tested, we can draw the conclusion that the memory efficiency, as well as the branch and occupancy efficiency are directly linked to the recorded instructions per clock – when these metrics are higher, the IPC value also raises. An interesting point of note however is that even in this problem, when all recorded metrics are very high, the IPC count is still some distance away from hypothetical ideal result of 1. This indicates even when running our model on problems that are structured to allow the model to run highly efficiently, it is still an inherently memory bound algorithm.

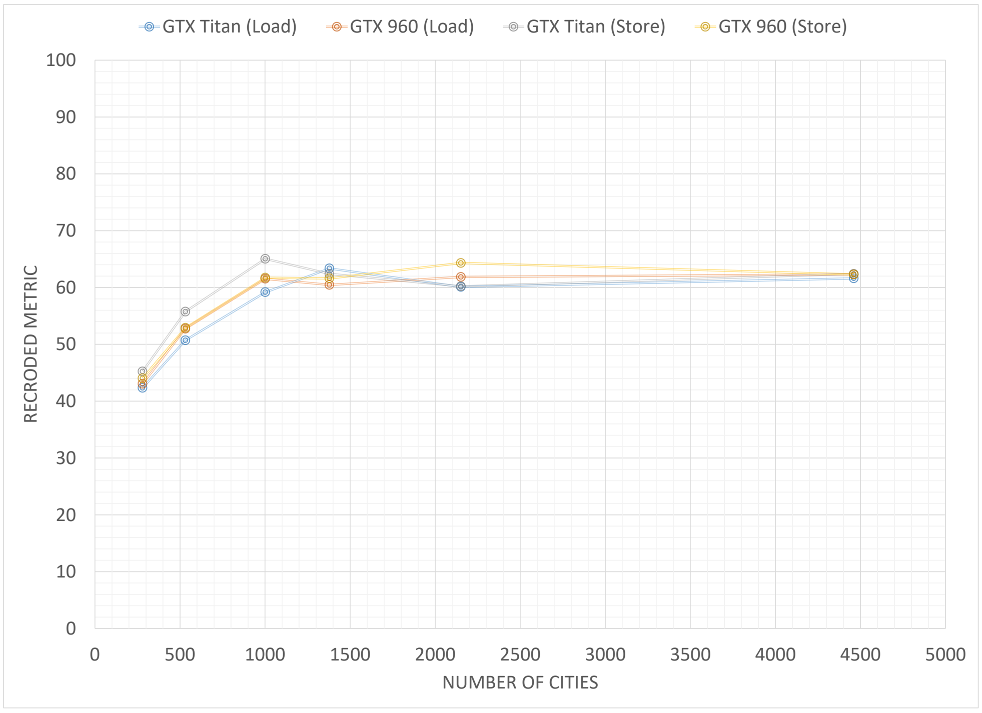


Fig. 6.15 Recorded global load and store efficiency whilst solving the travelling salesman problem, as the number of cities increases

6.1.5 The Travelling Salesman Problem

Finally we consider the travelling salesman problem. This problem is very different from all the previous problems implemented, to an extent it requires hard coded adaptations as detailed in Sect. 4.3. This also makes profiling challenging, as multiple sub-scoring grids are spawned through multiple kernels, it may not be clear what is causing any observed results. Therefore, we expect the results of this test vary considerably from what has been presented thus far.

Considering the global store and load efficiency value as presented in Fig. 6.15, the results clearly show a different trend to those observed during testing so far. Both store and load values are the lowest that have been recorded, dropping as low as 42% for the smaller instances, then levelling at roughly 63%. As well as the results being low overall, the trend of the metrics starting at a low value, then raising and levelling is also a new observation.

As aforementioned, determining the reason for these results is hard to isolate due to the nature of the multiple kernels being launched, and the profiler simply averaging results recorded across these. A key observation from the profiling results reported during testing is that the *maximum* value for the load and store metric recorded at any point during execution did in fact raise as high at 90%, more in line with previous results. Therefore, at points of the execution of the model efficiency was very high, although it averaged the observed low results.

Based on this we can draw a hypothesis on the reasons for the low results. Launching the sub kernels means increasingly smaller scoring grids are continually launched and executed, which will run less efficiently than the larger grids, as the size of the memory transactions to global memory are smaller. Therefore in the smaller problem sizes, the number of kernels spawned that execute small 'sub optimal' memory transactions is of a greater proportion compared to larger scoring grids, thus reducing the overall efficiency. We believe this is the reason for the trend of the metrics raising then reaching a plateau. The same hypothesis can be applied to the overall degradation in the efficiency, with the very small scoring grids required during execution bringing down the average efficiency metrics. We believe that the high recorded maximum value for both the load and store metrics is representative of the memory transactions taking place during the execution of the larger scoring grids, demonstrating the model performs more in line with previous results assuming the scoring grid is of an appropriate size.

Whilst the average result of 60% is lower than we have observed so far, it is still a strong result for the model when considering the complexity of the problem, and the large reliance on global memory operations, thus we consider this a positive result.

We now consider the other CUDA metrics recorded during the execution of the travelling salesman problem, which are given in Fig. 6.16. These results follow a very similar trend to the previous metrics recorded for the TSP problem, with a pattern of beginning low and

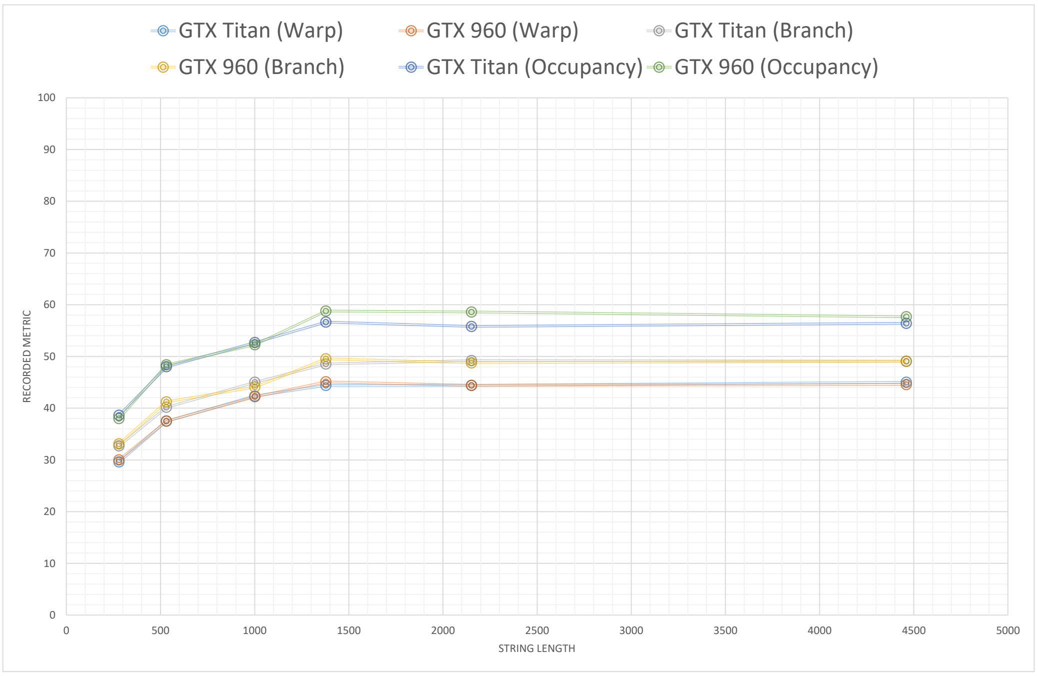


Fig. 6.16 Other recorded CUDA metrics whilst solving the travelling salesman problem, as the number of cities increases

raising to a point at which they level out as the problem size grows. However, in the case of these metrics compared to the previous, the rise of the initial slope is of a lower gradient, and it requires a larger problem sizes before the results begin to plateau.

Also, compared to the previous tests these results are lower still and again are the lowest observed metrics so far. This, however, is line with the overall pattern throughout testing where the second set of metrics concerning warp efficiency, branch efficiency, and occupancy are lower than the load and store metrics for the same test problem.

We believe that the reasons for the low values recorded for these metrics are the same as those which contributed to the low memory efficiency metrics. The warp and occupancy metrics are linked to the size of the memory operations that are taking place, and the size of the wave front. For example, if the size of the wave front is very small in one of the small sub kernels of the TSP, this will mean that a warp is spawned but not all threads are used impacting the efficiency. Similarly this will impact the occupancy, as multi-processors on the

Table 6.5 Average instructions per clock recorded during the performance testing of the travelling salesman problem

Number of cities	IPC
280	0.52
532	0.55
1002	0.55
1379	0.56
2152	0.55
4461	0.56

GPU are idling during execution. The branch efficiency will also be affected if small scoring grids are being solved, as more often the code within a warp will be required to take different paths as the bounds of the grid are reached.

It should be noted that whilst these metrics are low, they are not representative of the model more generally. The implementation of the TSP problem is highly specialised, and requires significant changes from the original model. However additional problems in the future could need to be implemented using similar techniques, so it is beneficial to have investigated the performance and associated drawbacks. We are still pleased with the metrics, as 60% efficiency is still a strong result and demonstrates a solid baseline for performance when multiple scoring grids are required, which can be built on in the future.

Instructions Per Clock

Finally we consider the instructions per clock recorded during the performance testing of the TSP problem. As with all testing for the TSP, we have little prior indication what to expect from these results, although based on testing thus far we expect them to be low.

Recorded IPC metrics are given in Table 6.5, and these show the unexpected result of values more in line with those observed in previous testing such as the LCS problem or the edit distance problem. We now seek to rationalise these surprising results. Whilst the implementation of the TSP problem is clearly highly memory bound, and the core definition to fill the cell is relatively simple which led us to expect a low IPC, there are many small memory transactions taking place in the small wave fronts of the small scoring grid. Therefore, whilst these memory transactions are inefficiently not using the full warp size, and reduce the associated metrics, they are small and take place quickly. We believe that the relatively high IPC metrics are due to the small memory transactions completing quickly, causing the algorithm to stall less during execution.

6.2 Comparative Benchmarks

We will now consider the performance of the model within the context of other published works. As aforementioned, some of the results presented here should not to be used for direct comparison, as re-implementing all similar algorithms and models is outside the scope of this thesis. In some instances we use other authors results directly which have been run on different hardware and software configurations. However providing comparisons is still useful, for example if our model were to outperform another algorithm with a speedup of many orders of magnitude, it is clear this is likely due to more than simple hardware differences. In other instances, where possible, we re-implement other authors work.

This section also provides simple empirical data for our algorithm such as run times and memory usage which has been not been presented thus far in the thesis, where we focused more on analysing CUDA metrics.

All comparative publications and algorithms used here are introduced in Sect. 5.3.3, with our rationale for selecting these.

6.2.1 Sequential CPU

First, we begin by comparing our model to the sequential CPU implementations available in the literature, or ones we have re-implemented for illustrative purposes. It should be apparent that we expect our model to perform significantly faster than classical single

Table 6.6 Comparative runtimes for sequential CPU based LCS algorithm, and our parallel model

	Runtime (s)				Mem (mb)
String Length	WM	NM	MM	Us	Us
5000	9.31	12.3	15.78	0.14	145
10000	18.61	24.6	31.56	0.28	147
15000	27.92	36.91	47.34	0.43	139
20000	37.22	49.21	63.12	0.57	140
25000	46.53	61.52	78.9	0.78	120
30000	55.83	73.82	94.68	0.82	130

core CPU implementations, however in the interest of evaluating the model thoroughly we systematically compare against a range of alternate solving methodologies.

The Longest Common Subsequence Problem

Beginning with the LCS problem we consider the algorithm presented in a survey paper from 2000 [11] where the authors re-implement three different diagonal based methods, similar in structure to our model. Due to the age of this paper we re-implement the same three diagonal based methods for local execution to provide a comparison on more modern hardware. The three sequential algorithms we compare against are the works of Wu, Nakatsu, and Miller [96, 67, 64], and we abbreviate these to WM, NM and MM respectively. This test will provide a clear comparison to what our proposed model offers compared to a similar algorithm running on the CPU. In these tests, both strings used during testing are of identical length, and the match percentage is fixed at 50%. Note the significantly reduced string sizes as the CPU implementation has no memory management and therefore the whole scoring grid must be maintained in system memory.

The results are given in Table 6.6, and shows the expected strong performance for our model, with our GPU implementation executing upto 115x faster than an equivalent CPU algorithm in the best case. An interesting point is the memory usage appears to drop as the size of the input string increases. This is due to the fact with our memory management less

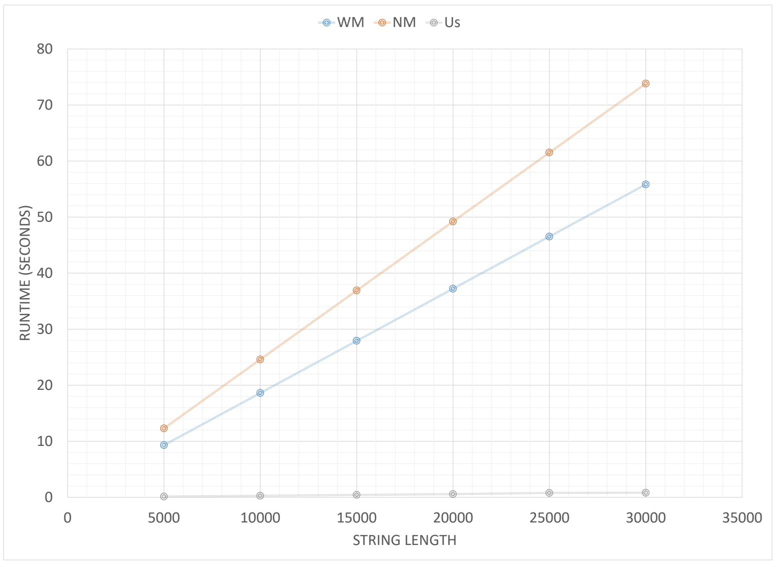


Fig. 6.17 Runtime for sequential CPU algorithms, compared to our proposed model when solving the longest common subsequence problem, as the size of the string grows

than 20 megabytes is required to be maintained on the GPU during such small scale testing, therefore any recorded memory values are only CUDA overheads.

Edit Distance Problem

Next we consider the edit distance problem. For this problem we re-implement the simple dynamic programming algorithm on the CPU. As with the previous tests, both strings are of the same size, and the size of the strings are considerably restricted due to memory limitations of the CPU version.

Table 6.7 Comparative runtimes for a sequential CPU based edit distance algorithms, and our parallel model

	Runtime (s)		Mem (mb)
String Length	CPU	Us	Us
5000	19.64	0.16	149
10000	39.28	0.32	147
15000	58.93	0.48	144
20000	78.57	0.64	148
25000	98.21	0.8	148
30000	117.86	0.97	146

Table 6.8 Comparative runtimes for a sequential CPU based manhattan distance problem, and our parallel model

	Runtime (s)		Mem (mb)
City Dimension	CPU	Us	Us
5000	20.63	0.64	148
10000	41.26	1.28	143
15000	61.89	1.92	142
20000	82.52	2.56	140
25000	103.16	3.21	147
30000	123.79	3.85	144

Unsurprisingly, the results for the Edit Distance problem (shown in Table 6.7) follow a similar pattern to the previous tests, where the GPU algorithm considerably outperforms the CPU equivalent by a large margin (a 121x speedup in the best case for our model compared to the CPU), using very little memory in the process.

Manhattan Tourist Problem

In the interest of completeness, testing of the Manhattan tourist problem is also carried out, although we expect very little deviation from the already observed pattern during testing due to the similarity of the problem structures. As with the previous test, we create a simple CPU based dynamic programming version, and limit the problem size as the whole scoring grid must be maintained.

In line with the previous testing, the observed GPU run-time in Table 6.8 is a fraction of the CPU runtime, and the memory usage for our model is minimal utilising a barely perceptible amount beyond the standard CUDA overhead.

Knapsack Problem

Next we consider the knapsack problem. As noted during the previous performance testing, the knapsack problem has a more challenging memory structure when running through the model.

Table 6.9 Comparative runtimes for a sequential CPU based 0/1 knapsack problem algorithm, and our parallel model

	Runtime (s)			Mem (mb)
Knapsack Capacity	CPU	PI	Us	Us
10000	15	9.88	2.34	193
20000	27.48	16.12	3.52	202
30000	39.66	21.18	4.49	213
40000	65.65	29.2	6.44	229
50000	101.56	38.14	9.82	240
60000	187.5	44.92	12.63	263
70000	290.91	51.13	15.92	271
80000	501.31	81.15	18.97	299
90000	740.57	126.56	29.17	301
100000	1258.26	160.34	31.14	310

As well as implementing a basic dynamic programming based algorithm on the CPU we also re-implement a dynamic programming algorithm from 1994 [77] (PI), which was re-investigated again in 2000 [61]. During the testing, the capacity of the knapsack was altered, and the size of the item set was fixed at 1% of the capacity. Items within the item set were generated using the same parameters as in the previous performance testing. Due to the structure of the scoring grid for this problem, we can test instances using the CPU of a similar size to that of the GPU – although the CPU version requires more memory, enough is available on the test hardware.

Results for the comparative tests are given in Table 6.9, and shown in Fig 6.18. The results, again, clearly demonstrate our model out performing the CPU implementation, with a runtime improvement of upto 40x in the best case. When running this problem we observe memory usage rising for the first time in the comparative testing as the problem size grows. However, memory usage is still minimal even as the problem size grows, for example only using 300mb when the capacity grows as high as 100,000. This is a demonstration of our memory model in action, showing that our approach to only keeping the needed iterations of the wave front in memory allows very large problem instances to be solved.

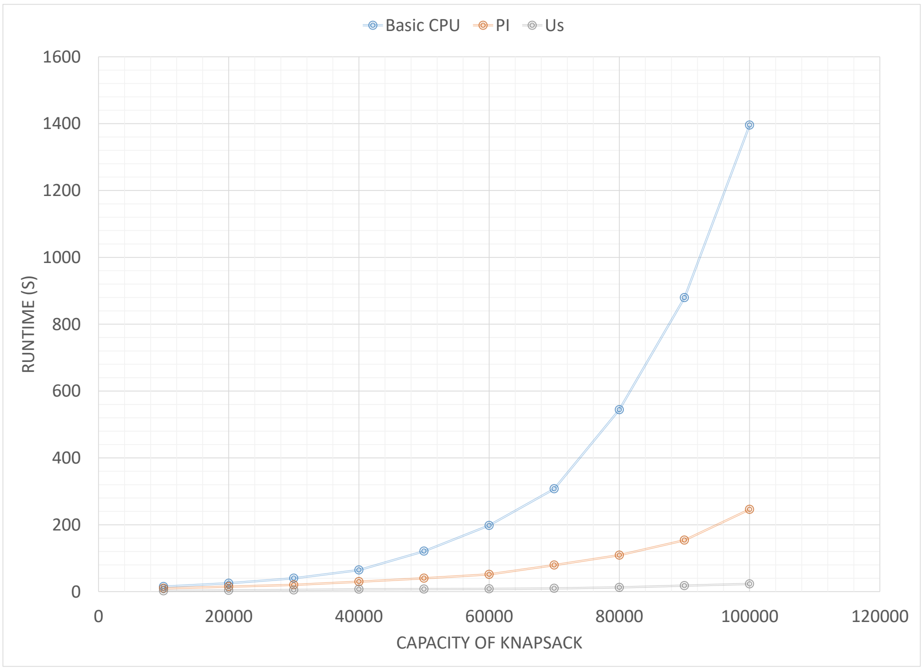


Fig. 6.18 Runtime of sequential CPU algorithms, compared to our parallel model when solving the 0/1 knapsack problem, as the capacity of the knapsack grows

Travelling Salesman Problem

The travelling salesman problem is considerably harder to test, as there is little focus in the literature on developing exact methods to solve it using only a single CPU core. Therefore we implement a CPU based method of our solving methodology, using multiple scoring grids. However, this is lacking any kind of memory management, and as such testing is limited only to very small instances.

Due to this we are required to select much smaller instances from the TSPLib for use during this phase of testing, and we select br17, fri26, bays29, ftv33, p43, hk48.

The results for the travelling salesman problem are given in Table 6.10 and Fig. 6.19. These results demonstrate our parallel implementation outperforming the CPU equivalent by upto 10x in the best case. Whilst in previous tests, our model has consistently performed more quickly than the equivalent CPU counterpart, this problem demonstrates the smallest speedup factor observed thus far. Also considering the memory we can see the TSP records the largest

Table 6.10 Comparative runtimes for a sequential CPU based travelling salesman problem solver, and our parallel model

	Runtime (s)		Mem (mb)
City Dimension	CPU	Us	Us
17	17.93	31.39	325
26	56.67	40.78	388
29	179.44	47.38	426
33	652.48	60.01	472
44	1852.66	117.64	613
48	2472.43	233.88	690

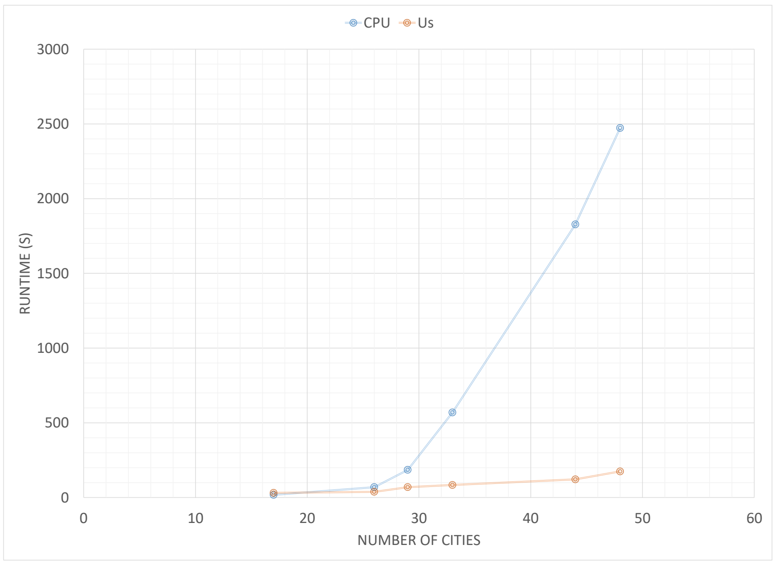


Fig. 6.19 Runtime for a sequential CPU algorithm, compared to our parallel model when solving the travelling salesman problem as the number of cities grows

results observed throughout comparative testing, using roughly 700MB for a 48 city instance. The equivalent problem running on the CPU based implementation required over 3GB of system memory, further demonstrating the effectiveness of the memory management in use our proposed model

6.2.2 Parallel CPU

The next phase of our comparative testing is concerned with comparing our implementation against parallel CPU based algorithms. This will serve to give a more representative performance comparison of the model against CPU based equivalents, as parallel algorithms are far more likely to be used in the practical settings, rather than classic sequential methods.

Longest Common Subsequence Problem

We begin by considering the longest common subsequence problem. For comparison we are using a relatively modern algorithm from 2006 [54] (KR), which solves chunks of the scoring grid, then moves on to solve others as dependencies become satisfied. The algorithm is built upon the earlier work of Crochemore [21], which is a bit-parallel approach. This paper was easy to re-implement as the work it details is based on batch sequential processing (BSP), and built upon the Oxford BSPlib toolbox, therefore we could quickly replicate the authors work.

Although the toolbox allows for deployment across a cluster style system, in line with previous testing we ran it locally on the single machine using MPI for local communication. All cores of the machine were used, including virtual as well as physical cores, meaning the CPU version was deployed on 12 cores. The length of the string was varied through testing, and the match percentage was fixed at 50%. The bit parallel representation of the data in the comparative algorithm means the CPU memory requirement has dropped allowing us to compare larger problem instances.

Table 6.11 Comparative runtimes for a parallel CPU based longest common subsequence problem solver, and our parallel model

	Runtime (s)		Mem (mb)
String Length	KR	Us	Us
25000	6.94	0.58	145
50000	11.20	1.05	148
75000	17.52	1.94	147
100000	25.87	2.39	148
125000	38.29	3.37	145
150000	56.39	4.18	145
175000	81.22	5.19	147
200000	117.90	6.78	149
225000	170.02	8.62	147
250000	245.07	11.3	149
275000	352.82	14.09	148
300000	507.22	17.62	148

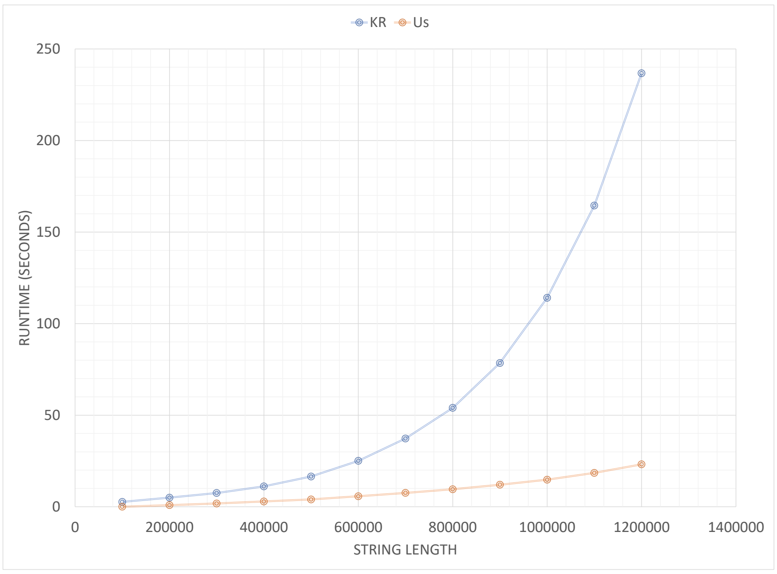


Fig. 6.20 Runtime for parallel CPU algorithms, compared to our parallel model when solving the longest common subequence problem as the string length grows

The results of the parallel CPU test are given in Fig. 6.20 and in Table 6.11. These show our model still comfortably out performing the parallel CPU version by a speedup factor of 28x, even when it is deployed across 12 cores. Another point to note is the memory usage of our model is still remaining constant as the problem size is growing. This is due to the fact in the implementation of this problem, only 3 iterations are required to be maintained on the GPU, and we are choosing not to maintain the entire scoring grid on the host.

We now continue our investigation into the performance of the parallel CPU algorithm and attempt to determine how many CPU cores would be required to match the performance of our GPU model. To achieve this we observe the performance of the KR algorithm using different numbers of cores to identify the additional runtime improvement as more cores are added, which allows us to extrapolate the number of cores needed to match the performance of our GPU model. This will only serve to provide a rough estimate as factors such as overheads, scaling and communication costs are not taking into account, and this assumes the run-time continues to drop linearly as more cores are added which is unlikely. Therefore this is an estimate of best case performance for the CPU algorithm. During these tests, the string length is fixed at 300,000, and the number of cores is varied between 1 and 6, so only the physical cores of the CPU are being utilised to give consistent results across testing.

The recorded values as the number of cores increase are presented in Fig. 6.21. Based on the observed trend in these results, we can assume that if the scaling continued in this manner, the CPU parallel version would require approximately 250 cores to match the performance of the GPU. This is a testament to the efficiency of our model, and also GPU programming more generally, where one discrete GPU card can deliver similar performance to a cluster sized CPU system.

As outlined in the testing methodology discussed in Sect. 5.3.3, due to the similarity of the problem structures of the LCS problem, the edit distance problem, and the Manhattan tourist problem, we do not perform comparative CPU parallel tests for each individual

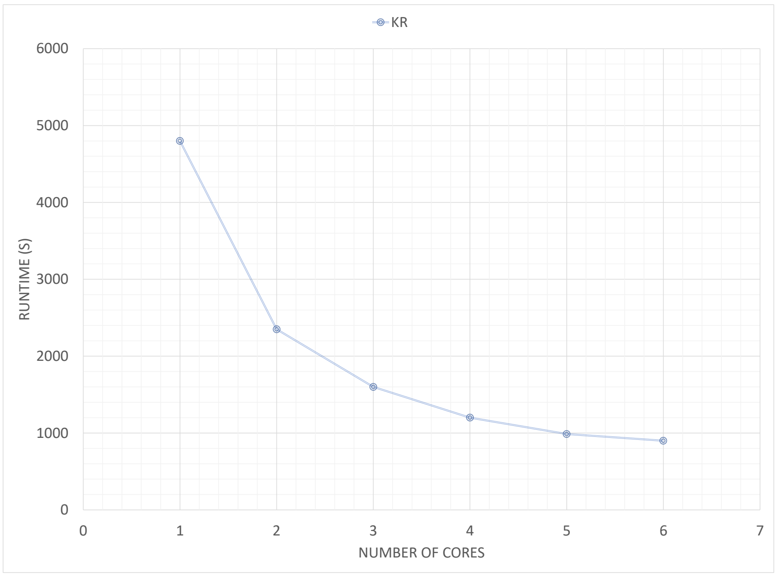


Fig. 6.21 Runtime for the parallel CPU based KR algorithm solving the longest common subsequence problem as the number of cores is increased

problem. Instead we use the results of the CPU parallel LCS tests as an indicator to the performance of these other problems. This assumption is valid, based on the similarity of the results observed throughout testing thus far.

The Knapsack Problem

Next, we consider the knapsack problem, comparing against the work of a recent review paper focused on parallel CPU based algorithms for solving the KSP [80]. The paper focuses on parallelising the basic knpasack dynamic programming recursion using two methods, the classic method of solving each row in parallel, and a parallel pipeline method proposed by Morales [66]. We compare against both these methods, and refer to them as CLA and MOR respectively. The paper also decomposes the scoring grid into blocks to improve the performance of each thread.

Due to the explicit pseudo-code provided in this paper it was easy to replicate the implementation of the algorithms. As with previous Knapsack testing we alter the capacity, and fix the size of the item set at 1% of the capacity.

Table 6.12 Runtimes for our model compared to two parallel CPU based knapsack problem solvers

	Runtime (s)			Mem (mb)
Capacity	CLA	MOR	Us	Us
10000	11.26	9.10	2.34	200
20000	16.03	11.53	2.66	215
30000	23.42	17.58	3.85	218
40000	31.45	20.23	5.02	241
50000	42.8	28.6	6.86	252
60000	60.54	36.51	10.31	261
70000	96.09	54.24	12.89	280
80000	110.22	74.76	18.24	310
90000	156.33	119.08	25.48	315
100000	204.29	159.81	35.52	364

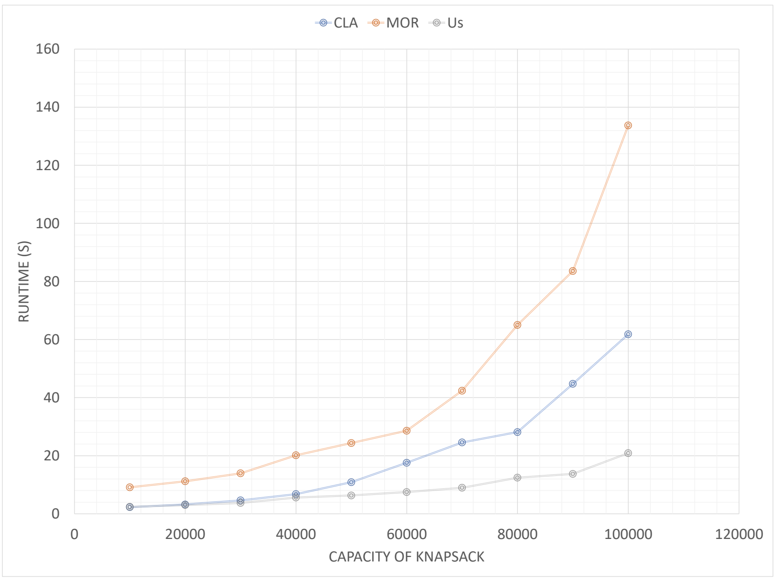


Fig. 6.22 Runtime for parallel CPU algorithms, compared to our proposed parallel model when solving the 0/1 knapsack problem, as the capacity of the knapsack grows

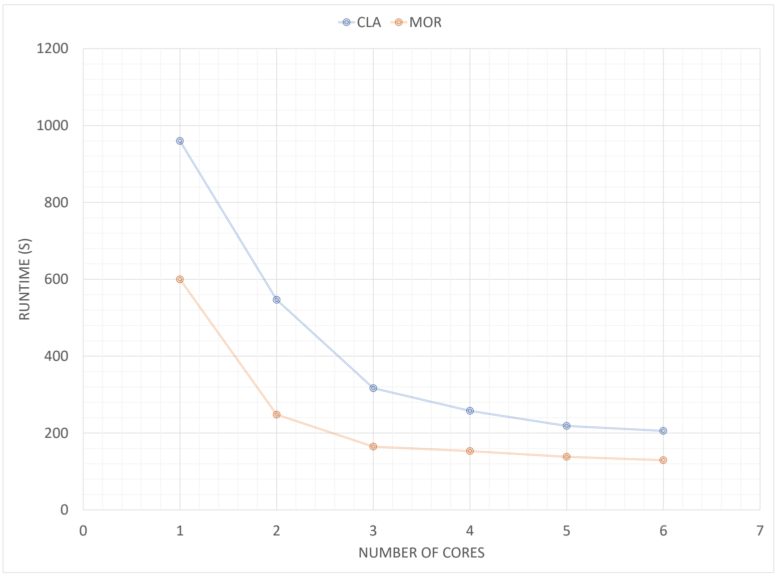


Fig. 6.23 Runtime for the parallel CPU based CLA and MOR algorithm solving the knapsack problem as the number of cores is increased

Results of these tests are shown in Fig. 6.22 and detailed in Table 6.12. In line with previous results our algorithm demonstrates a clear performance improvement over the competing parallel CPU algorithms, showing upto a 5x runtime improvement. Also, the memory usage remains low as observed in previous testing.

We next consider how many CPUs would be required when using these algorithms to provide comparative performance to that of our model. This is done using the same methodology as calculating the parallel CPU scaling for the longest common subsequence problem: by simply recording the performance as the number of cores used increases and extrapolating this value beyond the number of cores available. During these tests, the capacity of the Knapsack is fixed at 100,000 and the only parameter changing between tests is the number of cores the parallel CPU algorithm is running on.

The results of the scaling test are given in Fig. 6.23. Based on the results observed here, we can assume that if scaling continued in the same manner beyond 6 cores to *n* cores, the parallel CLA algorithm would require a system of approximately 85 cores to offer comparative performance to our algorithm, and the MOR algorithm would require a system of approximately 50 cores to achieve this. This result is surprising – a parallel CPU algorithm requires approximately 50 cores to offer similar performance to a GPU algorithm which is deployed across thousands of cores. Whilst it is common knowledge that, generally, fewer CPUs are required to provide performance similar to a GPU, this is a bigger gap than expected. However, this result is promising as it demonstrates that even when the CPU is running an efficient, parallel algorithm, with enough cores to offer similar performance, our model is still the considerably cheaper and more power efficient solution.

6.2.3 Parallel GPU

Finally we consider the performance of competing GPU algorithms. These comparisons are the most important that will be drawn, as they demonstrate the performance of our proposed model directly compared to modern algorithms targeting the same hardware platform. During the comparative GPU testing we seek to use test data that is larger than has been used thus far during the comparative benchmarks.

Longest Common Subsequence Problem

We begin by considering the longest common subsequence problem. Comparisons will be drawn against the 2008 algorithm of Kloetzli (KLO) [53] which is similar to our parallel model, as well the 2010 algorithm of Yang (YAN) et al. [99].

As with our model, Kloetzli also traverses through the scoring grid using a diagonal wave front, and transforms how the data is stored in memory such that each diagonal iteration can be accessed using a linear memory request. The scoring grid is broken down into small subgrids, allowing smaller sub-problems to be solved which are reconstructed into the final solution. A key difference between our model and this algorithm however is Klotezli stores the entire scoring grid on the GPU, and offers no solution to memory management as the size of the test data grows. Yang et al. similarly traverses the grid filling cells in parallel

Table 6.13 Runtimes for our model compared to two parallel GPU based longest common subsequence solvers

	Runtime (s)			Mem (mb)
Strnig Length	KLO	YAN	Us	Us
10000	3.66	1.78	0.1	145
20000	6.61	3.33	2.1	147
30000	10.2	5.28	9.66	146
40000	16.1	8.56	14.08	146
50000	24.23	12.43	19	145
60000	36.15	18.46	26.91	148
70000	53.52	26.9	35.79	149
80000	78.11	39.57	44.85	190
90000	112.55	57.59	53.88	212
100000	162.83	83.47	71.81	253
120000	235.09	120.32	91.96	359
110000	338.55	173.79	115.44	398

as dependencies become satisfied, but the dependencies of the algorithm are restructured such that larger proportions of the scoring grid can be filled simultaneously. This paper is interesting as the results provided by the author demonstrate it running faster than a traditional diagonal parallel method.

Both of these algorithms have been re-implemented by ourselves for use in an earlier publication [74], using the pseduo-code provided by the authors. During testing, the match length was fixed at 50% of the string, and the length of the string was varied. Through all the testing, the lengths of both strings was equal.

Results of these tests are shown in Fig. 6.24 and presented in Table 6.13. The results show a clear advantage for our model compared to the competing algorithms, with a runtime improvement of upto a factor 3x. Note however that for small problem instances, the YAN algorithm performs marginally quicker than our model and it is only as the problem size grows that our model has an advantage. We hypothesise this is due to the additional synchronisation required by our model between iterations of the wave front. Also, in these

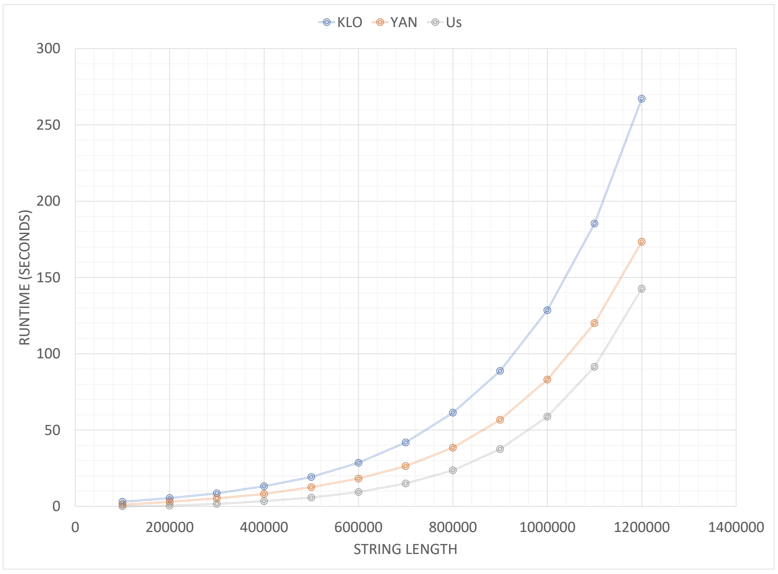


Fig. 6.24 recorded run-times for our our model when solving the longest common subsequence problem as the string length varies, compared to other GPU based algorithms

larger instances we observe the memory usage of our model beginning to rise for the first time during the testing of the longest common subsequence problem.

As with the previous testing, we elect to use the results of the longest common subsequence testing as an indicator of the performance of the edit distance problem, and for the Manhattan tourist problem.

The Knapsack Problem

To perform the comparative testing for the Knapsack Problem on the GPU we use the algorithm (BOY) described in the 2012 paper by Boyer et al. [14]. This paper demonstrates an algorithm tailored to solving the knapsack problem, showing multiple techniques to solve this problem efficiently using the GPU: it is a row parallel algorithm, which also has elements of bit parallel processing in order to further accelerate the algorithm. Also, it demonstrates compressing data in memory such that the GPU can solve larger problems, and the transfer time to retrieve data from the GPU is reduced.

Table 6.14 Comparative runtimes for parallel GPU based 0/1 knapsack algorithm compared to our model

	Runtime (s)		Mem (mb)
Capacity	BOY	Us	Us
50000	11.19	12.15	330
100000	14.58	14.5	429
150000	22.93	16.12	557
200000	26.69	18.53	725
250000	35.07	25.94	942
300000	51.51	38.25	1225
350000	64.73	46.21	1592
400000	79.91	70.7	2070
450000	120.81	89.82	2691
500000	184.62	123.1	3499

For comparative testing of this algorithm we kept the size of the item set fixed at 1% of the capacity, and the capacity was altered between tests. Due to the memory management offered by both algorithms being considered we can investigate considerably larger instance than previously.

Results of these tests are shown in Fig. 6.25 and shown in Table 6.14. Whilst the results show our model performs more quickly than the competing algorithm, this is only by a small margin. Also, the BOY runs more quickly when executing smaller instances. Although these results are close, this is a very encouraging result for our model, as it demonstrates that it can achieve performance that is quicker than a specialised GPU algorithm dedicated to solving the problem. Finally, we also note the memory usage is considerably higher than has been observed so far during testing, but this is to be expected as these are considerably larger test instances than has been used before.

The Travelling Salesman Problem

As with the previous comparative benchmarks, there is a lack of literature pertaining to the parallel implementation of the TSP in an exact manner on GPUs, so no comparative testing can be carried out.

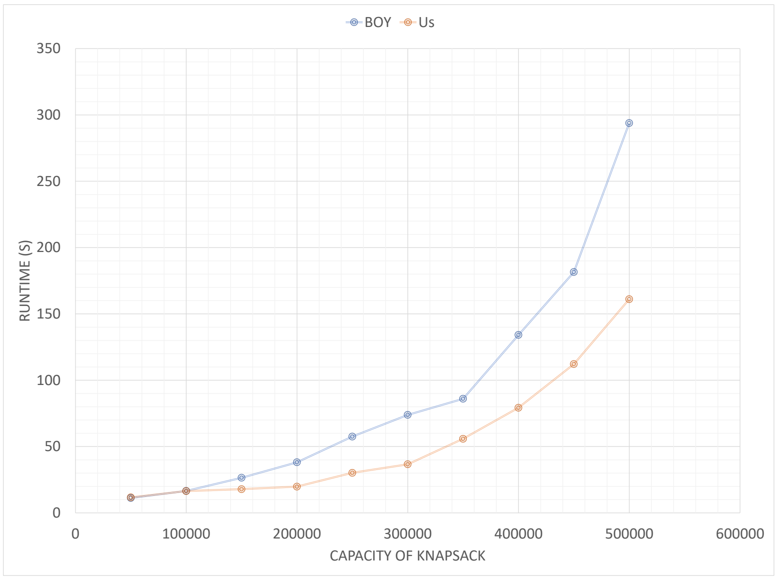


Fig. 6.25 Runtimes for our our model when solving the 0/1 knapsack problem as the capacity varies, compared to another GPU based algorithm

6.3 Key Findings

We will now identify the key points that have been observed during the testing of the model.

- The model was successful in allowing problems to be solved, using different dynamic programming algorithms.
- Global memory load and store metrics are consistently high, averaging above 85% across nearly all test instances.
- Warp divergence and observed occupancy were also consistently high, averaging above 75% across most test instances.
- The recorded IPC metric was lower than expected universally, indicating a high reliance on memory operations.
- Compared to sequential CPU implementations, a speedup was observed of upto a 121x run-time improvement. For all problems other than the travelling salesman problem, our model offered at least a 40x improvement of run-time.

- Our proposed model performed quicker than parallel CPU implementations, providing a speedup of up-to 28x compared to a 12 core CPU.
- Based on our findings, we hypothesise a CPU cluster of at *least* 50 cores would be required to offer similar performance to our GPU based model.
- Our proposed model performs twice as fast as some specialised GPU parallel algorithms, and offers performance comparable to others. Never does our model perform more slowly on average than competing GPU algorithms.

Summary

This chapter has analysed the performance of the generic parallel model when it is applied to the different problems that have been introduced in this thesis. The analysis has allowed conclusions to be drawn on the overall effectiveness of the model.

The model was evaluated through the implementation five different test problems recording a range of CUDA metrics, including memory efficiency metrics, divergence metrics, and instructions per clock. This allowed for an insight into how effectively the model is using the resources of the GPU, also considering hardware of different ages providing different amounts of CUDA cores.

Comparisons have been drawn against results from the literature to demonstrate what our model offers compared to existing solutions, and to better provide context to the contributions of our algorithm.

Chapter 7

Conclusion

Overview

This chapter presents the conclusions that have been drawn from the work detailed in this thesis. Over arching findings are presented followed by discussions around more detailed conclusions. Future extensions for the model described in this thesis are also discussed.

7.1 Introduction

The model that has been introduced in this thesis has been presented as a method that allows the mapping of dynamic programming based algorithms to the GPU quickly and easily, and enables a single paradigm to be used to solve a range of different problems. Through the use of the test problems examined in this thesis we have determined that considerable performance advantages are available through the use of our model, and GPU solutions to multiple problems can be implemented. However, not all problems are implemented. Also, the performance benefits offered by the model vary from problem to problem.

When the model is running on 'simple' problems such as the longest common subsequence problem, the edit distance problem or the Manhattan tourist problem, there are large performance improvements available – an average of over a 100x run-time improvement compared to serial CPU implementations. The recorded CUDA metrics when running these problems of an average 85% or higher demonstrates the hardware of the GPU is very effectively used. This is likely due to the fact these problems were the starting point for our model, and the original inspiration for the parallel structure.

However moving onto problems such as the travelling salesman problem, whilst implementation is possible and there is a 10x speedup offered by running the algorithm on the GPU, considerable adjustments are required to the implementation to support this. Also, memory is very much a limiting factor for this problem, as even with memory management techniques and discarding as much old data as possible, quickly large instances become unfeasible to solve. Furthermore, a solution to the all pairs shortest path problem cannot be implemented using our model.

The test case of the knapsack problem is very promising for our model however, as this is an implementation that could be described as less than ideal with the large dependence on previous iterations, and a less simple access pattern to memory compared to the LCS problem. However, this problem still shows an extremely strong 40x speed up compared to a serial CPU implementation, 5x compared to a parallel CPU implementation, and 1.5x compared to a GPU algorithm.

Code divergence metrics, such as warp efficiency and branch efficiency for our model are extremely promising, consistently about 75% in all but the TSP, and demonstrate the memory management of the model is highly effective, as well as the overall design pattern. A limiting factor of the model surprisingly proved to be the instructions per cycle metric, which indicates the GPU spends a lot of time waiting for memory based operations.

The toolbox developed during the course of this thesis is available to download from http://jfoconnell.net/research/phd. The bundle available for download contains a `Makefile` to build the software into the executable `cuda-generic`. Code to generate test problem instances can be found in `gen-data`, and reference CUDA kernel implementations to solve the test problems used in this work are found in `gpu/reference-implementations`.

7.2 Overall Model Performance

Compared to CPU implementations, as discussed, our GPU implementation outperforms sequential versions comprehensively across all test problems, including the test problems where our model is running at less than ideal efficiency such as the travelling salesman problem. Furthermore, our GPU version runs universally quicker than parallel CPU implementations designed to run on a multi-core processors. As the results section demonstrates, in the majority of cases we estimate it would require over 50 CPU cores to move close the speed of our GPU implementation.

A wide range of problems can be solved successfully using the presented model, and we have demonstrated the implementation of the longest common subsequence, edit distance, Manhattan tourist, knapsack, and travelling salesman problems. However, additional problems can be implemented through the definition of input files; for example, problems such as the uncapacitated facility location problem, and minimum delay problem can both be solved by only changing the input file, and dynamic programming case. These are but two examples, as any problem meeting the criteria defined in Sec. 4.3.3 can be successfully be solved. Also, should small adaptations be made to the model, as also outlined in Sec. 4.3.3, problems such as chain matrix multiplication, and the subset sum problem can be solved by the model.

Compared to existing GPU implementations from the literature, our model also performs favourably. Based on the results, the overarching pattern demonstrates that our model is highly competitive compared to the literature, although highly specialised algorithms targeting single problems can sometimes be as quick, or quicker in some problem instances.

As well as comparing to GPU implementations from the literature, we also implemented our model on different GPU hardware in order to draw conclusions as to how the model scales, and to better isolate how well the resources of the GPU are being used. Our proposed model demonstrably scales well moving from a CUDA environment providing roughly a thousand cores (the GTX 960), to an environment proving two and a half times as many (the GTX Titan), with the recorded CUDA metrics barely changing between the two systems.

7.3 Memory Management

Our memory management model is clearly very effective in terms of both restricting the amount of memory required on the GPU, as well as transferring memory asynchronously of computation through the use of CUDA streams.

Throughout our testing GPU memory rarely proved to be a limiting factor, with the exception of running the larger test cases for the travelling salesman problem. Instead host memory was still the limiting factor when we were trying to maintain the complete scoring grid. With our mechanism of only maintaining on the GPU the minimal amount of dependencies required for future iterations, in problems such as the edit distance problem where there is a very small amount of data to be maintained on the GPU, the largest problem that could be computed would be strings with lengths in the order of 100s of millions.

The asynchronous queue performed flawlessly throughout testing, never encountering issues such as queue overflows when the queue became too large to store data on the GPU which could not be transferred back to the host quickly enough. This proved to be an invaluable asset to the model, the amount of time spent transferring memory whilst computation is taking place, is nearly the whole time the model is running. Therefore if the transfer and execution had to happen independently it would double the run time of the execution of the model, if not more due to the additional synchronisation required.

Recorded metrics also support the memory model, showing very strong results for the global load efficiency, and global store efficiency metrics, which demonstrate the steps that we have taken to ensure that memory requests are aligned, and no memory transactions are required to branch. This is an important result for the model, as global memory transactions are the slowest CUDA operation available, therefore it is crucial these are running at a high efficiency value.

7.4 CUDA Efficiency

The recorded CUDA metrics relating to performance, and code efficiency, also show strong results. This is important as it shows how effectively the resources of the GPU are being used, and therefore how well the model will port between different GPUs with different hardware configurations, as well as how well it is likely to scale when future hardware is released.

The branch efficiency and warp efficiency metrics are generally recorded at 75% or higher, although they are dictated to a small degree by the problem being considered. Whilst we ensure that memory requests to the previous iterations stored locally on the GPU are optimised, and the data is padded, we do not have the same control over the test data that is available globally, nor the pattern in which this is accessed.

The achieved occupancy is also very high, as this metric is linked to the previous two metrics of branch and warp efficiency, as well as to the block size used. Whilst the block size was hard coded and is selected by the programmer based on the specific hardware, rather than being set by the model, this metric still demonstrates that should an appropriate block size be set, the model is capable of taking full advantage of the resources available.

The only slightly negative metric that was recorded was the instructions per clock (IPC) metric. This is somewhat harder to isolate the reasons why, as this is linked to the generated machine level instructions that actually run on the GPU.

The GPU has theoretical maximum of 2 instructions per clock when using 64 bit data. In our implementation we use 32 bit data, therefore we have a theoretical limit of 1. The recorded results show the model struggles to attain this value almost universally. Firstly, it should be noted that the maximum IPC value of 1 is theoretical, and would require a specially crafted, highly CPU bound implementation to reach this. And secondly, we believe the low IPC value is due to the memory bound nature of our model, where clock cycles are used loading and storing data from and to the previous diagonals currently residing on the GPU, rather than performing CPU operations.

7.5 File Format Effectiveness

The support for implementing methods to solve different problems quickly and easily was a big success of the presented model, demonstrating how alternate problems can be quickly solved on the GPU with only the minimal amount of programming and CUDA knowledge. There were some limitations of this, for example the dynamic programming definition obviously needs to be present at compile time which can limit usability. Also as mentioned there are some problems that could not be represented using the input file, however this is more a limitation of the model as a whole rather than the specific format of the input file.

We believe our approach of leaving the core definition of the function that is to be run on the GPU blank, and furnishing it with information about the current cell that is being filled, requiring the user to only input information on how this cell is filled is a unique concept.

There is naturally some overhead associated with using an approach that is applicable to multiple problems, the primary complaint being that it is unlikely to perform as quickly as a specialised implementation. Also, certain additional overheads are present in the implementation, such as having to translate the indices every time a memory request happens to hide the complexity of the memory pattern from the user. However, as demonstrated the model still performs exceptionally well, and such small overheads are acceptable in order to allow the model to be accessible to all as was the original aim.

7.6 Future Work

We will now discuss future works which could take place to improve or extend the model, and describe the direction in which the research could be taken in the future.

Firstly, a level of auto tuning and auto optimisation would greatly aid the algorithm. As a contribution of the model is the fact it is generic and supports multiple problems, the obvious extension would be that it changes the underlying CUDA parameters automatically to optimal values for the hardware that it is running on seamlessly without input from the user. This should be a quick adjustment to make, as there have been a collection of works relating to this issue, however it would require the algorithm to run for a short period of time initially to monitor how much memory it needs and how it operates before it tunes these values. Should this be implemented though, performance could be slightly improved, without any additional expertise required from the user.

The next logical extension would be to enable the model to take advantage of multiple GPUs. This also should be a fairly easy extension to implement as CUDA now offers programming constructs that allow the deployment of code on multiple GPUs easily and quickly. Care would have to be taken to ensure that managing the memory accesses for different devices does not in fact slow the model down more than the increased core count would speed up execution, and it is likely steps would have to be taken to slightly alter the memory pattern if multiple GPUs were being targeted. However, this would add a huge performance benefit to the model, and furthermore would potentially allow it to run in large scale multi-GPU clusters.

Beyond these small implementation based improvements, longer term the research should be taken in a direction that allows a wider range of problems to be implemented, that have more complex dependency access patterns – for example the all pairs shortest path problem. We would suggest the starting point for such research would be to consider dividing the scoring grid down into smaller blocks in some way, and storing multiple different sub blocks in memory, potentially giving the wave front access to an increased number of dependencies at different locations around the scoring grid. This will then cause issues with increased memory usage on the GPU and will most likely require a considerably different approach to memory management. However this would be a very beneficial improvement to the model, as allowing the user to input a wider range of problems can only be beneficial.

References

[1] Aarts, E. and Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Inc., New York, NY, USA.

[2] Almasi, G. S. and Gottlieb, A. (1989). *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

[3] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410.

[4] Apostolico, A. (1997). *Pattern Matching Algorithms*. Oxford University Press, Oxford, UK.

[5] Apostolico, A., Atallah, M. J., Larmore, L. L., and McFaddin, S. (1990). Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988.

[6] Azencott, R. (1992). *Simulated annealing: parallelization techniques*, volume 27. Wiley-Interscience.

[7] Bellman, R. E. (1962). Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63.

[8] Bellman, R. E. (2003). *Dynamic Programming*. Dover Publications.

[9] Bellman, R. E. and Dreyfus, S. E. (1962). Applied dynamic programming. Technical Report R-352-PR, RAND Corporation.

[10] Berger, K.-E. and Galea, F. (2013). An efficient parallelization strategy for dynamic programming on GPU. In *Proceedings of the 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 1797–1806.

[11] Bergroth, L., Hakonen, H., and Raita, T. (2000). A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval*, SPIRE 2000, pages 39–48. IEEE.

[12] Bondhugula, U., Devulapalli, A., Fernando, J., Wyckoff, P., and Sadayappan, P. (2006). Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium. IPDPS 2006.*

[13] Boyer, V., Baz, D. E., and Elkihel, M. (2011). Dense dynamic programming on multi GPU. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:545–551.

[14] Boyer, V., Baz, D. E., and Elkihel, M. (2012). Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1):42–47. Special Issue on Knapsack Problems and Applications.

[15] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O. (2010). State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33.

[16] Casti, J., Richardson, M., and Larson, R. (1973). Dynamic programming and parallel computers. *Journal of Optimization Theory and Applications*, 12(4):423–438.

[17] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., and Skadron, K. (2008). A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380.

[18] Chen, G.-H. and Jang, J.-H. (1992). An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing*, 18(7):811–821.

[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, 2 edition.

[20] Cotta, C. and Troya, J. (1998). A hybrid genetic algorithm for the 0-1 multiple knapsack problem. In *Artificial Neural Nets and Genetic Algorithms*, pages 250–254. Springer Vienna.

[21] Crochemore, M., Iliopoulos, C. S., Pinzon, Y. J., and Reid, J. F. (2001). A fast and practical bit-vector algorithm for the longest common subsequence problem. *Letters on Information Processing*, 80(6):279–285.

[22] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2006). *Algorithms*. McGraw-Hill Higher Education.

[23] Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., and Lavenier, D. (2014). Efficient multi-GPU computation of all-pairs shortest paths. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, pages 360–369.

[24] Dreyfus, S. E. (2002). Richard Bellman on the birth of dynamic programming. *Operational Research*, 50(1):48–51.

[25] Eckstein, J., Phillips, C. A., and Hart, W. E. (2001). Pico: An object-oriented framework for parallel branch and bound. In Dan Butnariu, Y. C. and Reich, S., editors, *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, volume 8 of *Studies in Computational Mathematics*, pages 219–265. Elsevier.

[26] Edmonds, P., Chu, E., and George, A. (1993). Dynamic programming on a shared-memory multiprocessor. *Parallel Computing*, 19(1):9–22.

[27] Ferreiro, A., García, J., López-Salas, J. G., and Vázquez, C. (2013). An efficient implementation of parallel simulated annealing algorithm in GPUs. *Journal of Global Optimization*, 57(3):863–890.

[28] Fiechter, C.-N. (1994). A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, 51(3):243–267.

[29] Floyd, R. W. (1962). Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345.

[30] Galil, Z. and Park, K. (1994). Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2):213–222.

[31] Garland, M., Grand, S. L., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., and Volkov, V. (2008). Parallel computing experiences with CUDA. *IEEE Micro*, 28(4):13–27.

[32] Gendreau, M., Guertin, F., Potvin, J.-Y., and Taillard, E. (1999). Parallel tabu search for real-time vehicle routing and dispatching. *Transportation science*, 33(4):381–390.

[33] Gerasch, T. E. (1991). A parallel approximation algorithm for 0/1 knapsack. In Feng, T.-Y., editor, *Proceedings 20th International Conference Parallel Processing*, volume 3.

[34] Gilmore, P. and Gomory, R. E. (1965). Multistage cutting stock problems of two and more dimensions. *Operations research*, 13(1):94–120.

[35] Gilmore, P. C. and Gomory, R. E. (1963). A linear programming approach to the cutting stock problem - part II. *Operations research*, 11(6):863–888.

[36] Glover, F. (1989). Tabu search – part I. *ORSA Journal on computing*, 1(3):190–206.

[37] Glover, F. (1990). Tabu search – part II. *ORSA Journal on computing*, 2(1):4–32.

[38] Glover, F., Laguna, M., Taillard, E., and De Werra, D. (1993). *Tabu search*. Baltzer Basel.

[39] Goldman, A. and Trystram, D. (2004). An efficient parallel algorithm for solving the knapsack problem on hypercubes. *Journal of Parallel and Distributed Computing*, 64(11):1213–1222.

[40] Goldreich, O. (2008). *Computational Complexity: A Conceptual Perspective*. Cambridge Press.

[41] Harish, P. and Narayanan, P. J. (2007). *Proceedings of the 14th International Conference on High Performance Computing*, chapter Accelerating Large Graph Algorithms on the GPU Using CUDA, pages 197–208. Springer Berlin Heidelberg, Berlin, Heidelberg.

[42] Harris, M. (2007). Optimizing CUDA. In *Conference for High Performance Computing, Networking, Storage and Analysis*. NVIDIA Developer Technology.

[43] Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675.

[44] Horowitz, E. and Sahni, S. (1974). Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292.

[45] Ibarra, O. H. and Kim, C. E. (1975). Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468.

[46] Janiak, A., Janiak, W., and Lichtenstein, M. (2008). Tabu search on gpu. *Journal of Universal Computer Science*, 14(14):2416–2427.

[47] Jansen, T. (1998). Introduction to the theory of complexity and approximation algorithms. In Mayr, E. W., Jurgen Promel, H., and Steger, A., editors, *Lectures on Proof Verification and Approximation Algorithms*, volume 1367 of *Lecture Notes in Computer Science*, pages 5–28. Springer Berlin Heidelberg.

[48] Jenq, J.-F. and Sahni, S. (1987). All pairs shortest paths on a hypercube multiprocessor. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 713–716.

[49] Jones, N. C. and Pevzner, P. (2004). *An Introduction to Bioinformatics Algorithms*. MIT Press.

[50] Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Upper Saddle River, NJ, USA, 2 edition.

[51] Katz, G. J. and Kider, Jr, J. T. (2008). All-pairs shortest-paths for large graphs on the GPU. In Luebke, D. and Owens, J. D., editors, *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 47–55. Eurographics Association.

[52] Kirk, D. (2007). Nvidia CUDA software and GPU parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 103–104, New York, NY, USA. ACM.

[53] Kloetzli, J., Strege, B., Decker, J., and Olano, M. (2008). Parallel longest common subsequence using graphics hardware. In Favre, J. M. and Ma, K.-L., editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association.

[54] Krusche, P. and Tiskin, A. (2006). Efficient longest common subsequence computation using bulk-synchronous parallelism. In Gavrilova, M., Gervasi, O., Kumar, V., Tan, C., Taniar, D., Lagana, A., Mun, Y., and Choo, H., editors, *Computational Science and Its Applications - ICCSA 2006*, volume 3984 of *Lecture Notes in Computer Science*, pages 165–174. Springer Berlin Heidelberg.

[55] Laarhoven, P. J. M. and Aarts, E. H. L., editors (1987). *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, Norwell, MA, USA.

[56] Lee, J., Shragowitz, E., and Sahni, S. (1988). A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, 5(4):438–456.

[57] Lin, J. and Storer, J. A. (1991). Processor-efficient hypercube algorithms for the knapsack problem. *Journal of Parallel and Distributed Computing*, 13(3):332–337.

[58] Lou, D.-C. and Chang, C.-C. (1997). A parallel two-list algorithm for the knapsack problem. *Parallel Computing*, 22(14):1985–1996.

[59] Lu, M. and Lin, H. (1994). Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848.

[60] Maier, D. (1978). The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336.

[61] Martello, S., Pisinger, D., and Toth, P. (2000). New trends in exact algorithms for the 0–1 knapsack problem. *European Journal of Operational Research*, 123(2):325 – 332.

[62] Merkle, R. and Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5):525–530.

[63] Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960). Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329.

[64] Miller, W. and Myers, E. W. (1985). A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040.

[65] Mittal, S. and Vetter, J. S. (2015). A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35.

[66] Morales, D., Roda, J., Almeida, F., Rodríguez, C., and García, F. (1995). Integral knapsack problems: Parallel algorithms and their implementations on distributed systems. In *Proceedings of the 9th International Conference on Supercomputing*, ICS ’95, pages 218–226, New York, NY, USA. ACM.

[67] Nakatsu, N., Kambayashi, Y., and Yajima, S. (1982). A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18(2):171–179.

[68] Nandan Babu, K. and Saxena, S. (1997). Parallel algorithms for the longest common subsequence problem. In *Proceedings of the Fourth International Conference on High-Performance Computing*, pages 120–125. IEEE.

[69] Niar, S. and Freville, A. (1997). A parallel tabu search algorithm for the 0-1 multidimensional knapsack problem. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 512–516.

[70] Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, 6(2):40–53.

[71] NVIDIA (2015a). *CUDA C Programming Guide*. NVIDIA, Santa Clara, California, 7.0 edition.

[72] NVIDIA (2015b). *CUDA Occupancy Calculator*. NVIDIA, Santa Clara, California.

[73] NVIDIA (2015c). *Profiler User’s Guide*. NVIDIA, Santa Clara, California.

[74] O’Connell, J. F. and Mumford, C. L. (2014). An exact dynamic programming based method to solve optimisation problems using GPUs. In *Proceedings of the Second International Symposium on Computing and Networking (CANDAR)*, pages 347–353. IEEE.

[75] Ozsoy, A., Chauhan, A., and Swany, M. (2013). Achieving teracups on longest common subsequence problem using GPGPUs. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems (ICPADS)*, pages 69–77. IEEE.

[76] Pearson, W. R. and Lipman, D. J. (1988). Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448.

[77] Pisinger, D. (1994). A minimal algorithm for the 0-1 knapsack problem. *Journal of Operations Research*, 45:758–767.

[78] Pisinger, D. (1995). *Algorithms for Knapsack Problems*. PhD thesis, Department of Computer Science, University of Copenhagen.

[79] Pospichal, P., Jaros, J., and Schwarz, J. (2010). Parallel genetic algorithm on the CUDA architecture. In Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekart, A., Esparcia-Alcazar, A., Goh, C.-K., Merelo, J., Neri, F., Preub, M., Togelius, J., and Yannakakis, G. N., editors, *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 442–451. Springer Berlin Heidelberg.

[80] Rashid, H., Novoa, C., and Qasem, A. (2010). An evaluation of parallel knapsack algorithms on multicore architectures. In *Proceedings of the 2010 International Conference on Scientific Computing*.

[81] Reinelt, G. (1991). TSPLIB- a traveling salesman problem library. *ORSA Journal of Computing*, 3(4):376–384.

[82] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP.

[83] Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison Wesley.

[84] Shi, X., Li, C., Wang, X., and Li, K. (2009). A practical approach of curved ray prestack kirchhoff time migration on gpgpu. In *Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, APPT '09, pages 165–176, Berlin, Heidelberg. Springer-Verlag.

[85] Stivala, A. D., Stuckey, P. J., and Wirth, A. I. (2010). Fast and accurate protein substructure searching with simulated annealing and GPUs. *BMC bioinformatics*, 11(1):1.

[86] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73.

[87] Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62.

[88] Szu, H. and Hartley, R. (1987). Fast simulated annealing. *Physics Letters A*, 122(3):157 – 162.

[89] Tan, G., Sun, N., and Gao, G. R. (2007). A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 135–144, New York, NY, USA. ACM.

[90] Toth, P. (1980). Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, 25(1):29–45.

[91] Vazirani, V. V. (2001). *Approximation Algorithms*. Springer Berlin Heidelberg.

[92] Venkataraman, G., Sahni, S., and Mukhopadhyaya, S. (2003). A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics*, 8.

[93] Viswanathan, V., Huang, S.-H. S., and Liu, H. (1990). Parallel dynamic programming. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 497–500.

[94] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

[95] Wu, C.-C., Ke, J.-Y., Lin, H., and chun Feng, W. (2011). Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism. In *Proceedings of the 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 96–103.

[96] Wu, S., Manber, U., Myers, G., and Miller, W. (1990). An O(NP) sequence comparison algorithm. *Letters on Information Processing*, 35(6):317–323.

[97] Xu, X., Chen, L., Pan, Y., and He, P. (2005). Fast parallel algorithms for the longest common subsequence problem using an optical bus. In Gervasi, O., Gavrilova, M., Kumar, V., Lagana, A., Lee, H., Mun, Y., Taniar, D., and Tan, C., editors, *Computational Science and Its Applications - ICCSA 2005*, volume 3482 of *Lecture Notes in Computer Science*, pages 338–348. Springer Berlin Heidelberg.

[98] Yang, C.-T., Huang, C.-L., and Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1):266–269.

[99] Yang, J., Xu, Y., and Shang, Y. (2010). An efficient parallel algorithm for longest common subsequence problem on GPUs. In Ao, S. I., Gelman, L., Hukins, D. W. L., Hunter, A., and Korsunsky, A. M., editors, *Proceedings of the World Congress on Engineering*, volume 1, pages 499–504. International Association of Engineers, Newswood Limited.

[100] Yu, Q., Chen, C., and Pan, Z. (2005). Parallel genetic algorithms on programmable graphics hardware. In Wang, L., Chen, K., and Ong, Y. S., editors, *Advances in Natural Computation*, volume 3612 of *Lecture Notes in Computer Science*, pages 1051–1059. Springer Berlin Heidelberg.