An Exact Dynamic Programming Based Method to Solve Optimisation Problems Using GPUs

Jonathan F. O'Connell and Christine L. Mumford
School of Computer Science & Informatics
Cardiff University
Cardiff, United Kingdom
Email: {oconnellj1, mumfordcl}@cardiff.ac.uk

***Abstract*— In this paper we present a dynamic programming based technique that is suitable for providing exact solutions to a subset of optimisation problems, using general purpose GPU computing. The primary feature of this model is to efficiently use the computational and memory resources of the GPU, whilst still remaining abstract enough to allow implementation on a variety of problems. Secondly, as exact dynamic programming methods are often limited by memory complexity, great consideration has been given to reducing this constraint, allowing large scale problems to be solved. To demonstrate the effectiveness of the proposed model we test it against three problems; the 0-1 knapsack problem, the longest common subsequence problem, and the travelling salesman problem.**

I. Introduction

Optimisation problems involve finding the best possible solution to a problem by adjusting sets of variables with constraints, and they are prevalent in many sectors of both industry and academia as they allow easy mathematical representations of real world problems [1]. The complexity associated with solving optimisation problems grows extremely quickly as more variables are added, and the problems soon become essentially impossible to solve within an acceptable time frame.

The most common method of solving optimisation problems is through the use of heuristics. Heuristics attempt to use educated guesses to help navigate through the search space, allowing an acceptable answer to be produced within a fraction of the time. By definition, heuristics are not guaranteed to find the global optima to the problem as they are not an exact solving method. Dynamic programming (DP) based techniques provide exact solutions to a subset of optimisation problems, considerably quicker than naïve brute force based methods. Unfortunately, due to high computational and memory complexities they are unsuitable for large scale problem instances [2].

GPGPU (General Purpose computing on Graphical Processing Units) has increased in availability considerably in recent years, through languages such as CUDA and OpenCL. This allows normal desktop computers to run programs that use the resource of GPUs, which have core counts in the thousands.

Against this background we propose a GPU based model suitable for providing exact answers to optimisation problems that can be solved through dynamic programming based methods. The proposed model minimises memory complexity, whilst using thousands of cores to cope with the high computational complexity, allowing large scale problems to be solved. It is also designed to be a high level, semi-abstract model. This allows for implementation on different problems with only a minimum of alteration, but it is still able to be accelerated through problem specific knowledge. The objective of this paper is to demonstrate the effectiveness of the proposed model against some common optimisation problems.

A. Primary Contributions

The primary contributions offered by the model described in this paper are:

- Extrapolation and enhancement of a parallel paradigm to allow applicability to a range of different optimisation problems through dynamic programming techniques.
- Development of a memory structure to allow the efficient management of memory which maintains only the minimum amount of data on the GPU, and transfers completed data asynchronously to the CPU meaning execution never pauses. This allows very large scale problem instances, that previously could not fit in memory, to be solved.
- Provides the ability to solve large scale optimisation problems *exactly*. Previously such problem instances relied on inexact solving methodologies such as metaheuristics.

II. Optimisation Problems

A. Outline

An optimisation problem involves finding the best possible solution from a set of feasible solutions. In its most simple form, an optimisation problem can be defined as:

$$\begin{aligned} &\min_{x} f(x) && \\ \text{subject to: } &g_i(x) \leq 0, && i = 1, ..., m \\ &h_i(x) = 0, && i = 1, ..., p \end{aligned} \tag{1}$$

where $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ is the objective function to be minimised over x. $g_i(x) \leq 0$, are the inequality constraints and $h_i(x) = 0$ are the equality constraints.

B. Dynamic Programming

Dynamic programming is a term used to describe a method of programming in which a large or complex problem is broken down into smaller subproblems, which are quicker to solve and can be reconstructed into the final solution. As discussed, dynamic programming methods guarantee optimality as well as an exact answer. Within the context of optimisation problems, this means *the* optimal solution to the problem is guaranteed, but this leads to high computational and memory complexity.

C. Example Problems & Associated DP Solving Methodology

1) Longest Common Subsequence Problem: The longest common subsequence (LCS) problem is concerned with identifying the longest subsequence present in a set of multiple input strings. This problem is at the core of many bioinformatics and pattern matching algorithms.

String $C = c_1, c_2, ..., c_p$ is a subsequence of string $A = a_1, a_2, ...a_m$ if there is a mapping $F : \{1, 2, ..., p\} \rightarrow \{1, 2, ..., k\}$ such that $F(i) = k$ iff $c_i = a_k$ and F is a strictly increasing function. 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 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

This can be solved through DP by constructing a scoring matrix S of size n by m in which S_{ij} (where $1 \leq i \leq n, 1 \leq j \leq m$) maintains the longest common subsequence for substrings $a_1, a_2, ...a_i$ and $b_1, b_2, ..., b_j$; therefore the LCS length can be found at $S_{n,m}$. These values are filled using the following function:

$$S_{ij} = \begin{cases} 0 & i \text{ or } j = 0 \\ S_{i-1,j-1} & a_i = b_j \\ \max\left(S_{i-1,j}, S_{i,j-1}\right) & \text{otherwise} \end{cases} \tag{2}$$

2) 0-1 Knapsack Problem: The knapsack problem is a classic algorithm in combinatorial optimisation. Given a set of items with an integer mass and value, select the most profitable set of items whilst not exceeding a defined weight constraint. Although this problem is simplistic in nature, when extended it forms the basis of many packing and shipping algorithms.

The 0-1 Knapsack Problem restricts the number of times each item can be selected to zero or one. Let there be a set of items, $z_1, z_2, ..., z_n$ 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 $W \in \mathbb{N}_+^*$.

$$\begin{aligned} &\max \sum_{i=1}^{n} v_i x_i \\ &\text{subject to: } \sum_{i=1}^{n} w_i x_i \leq W, \qquad x_i \in \{0, 1\} \end{aligned} \tag{3}$$

This maximises the sum of the value of the items selected, whilst ensuring that the weight constraint, W, is not violated.

To solve this through DP, a scoring grid S must be created of size n by W in which S_{ij} (where $1 \leq i \leq n, 1 \leq j \leq W$) defines the maximum value that can be attained with weight less than or equal to W using items up to i; therefore the maximum value can be found at S_{nW}. The values in the grid are filled using the following function:

$$S_{ij} = \begin{cases} 0 & i = 0 \\ S_{i-1,j} & w_i > j \\ \max\left(S_{i-1,j}, S_{i-1,j-w_i} + v_i\right) & w_i \leq j \end{cases} \tag{4}$$

3) The Travelling Salesman Problem: The travelling salesman problem (TSP) is a well established problem within the field of computer science. Given a set of cities, labelled $1, 2, ..., n$, the goal is to find the shortest route that allows each city to be visited exactly once and the route to start and end at the same point. It has a huge computational complexity making exact solving impossible; however, it is at the core of many vehicle routing algorithms.

For an n city problem x_{ij} is defined as:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

Let u_i be a temporary variable and $c_{i,j}$ be the cost of moving between cities i and j. This problem can then be defined as an integer linear program [3]:

$$\begin{aligned} &\min \sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij} \\ &\text{subject to: } 0 \leq x_{ij} \in \{0, 1\} \leq 1 && i, j = 0, ..., n \\ &\qquad u_i \in \mathbb{Z} && i = 0, ..., n \\ &\qquad \sum_{i=0, i \neq j}^{n} x_{ij} = 1 && j = 0, ..., n \\ &\qquad \sum_{j=0, j \neq i}^{n} x_{ij} = 1 && i = 0, ..., n \\ &\qquad u_i - u_j + n x_{ij} \leq n - 1 && 1 \leq i \neq j \leq n \end{aligned} \tag{5}$$

The first set of equalities ensure that each city can only be arrived at from exactly one other city. The second set of equalities requires that from each city there is a departure to exactly one other city. Finally, the last constraint ensures that there is only a single tour covering all cities, and there cannot be multiple, simultaneous disjoint tours.

To represent this through DP, assuming city x as a start point: for every other vertex i we find the minimum cost path with x as the starting point, i as the ending point, and all vertices appearing exactly once. Let the cost of this path be p_i. Therefore the cost of the cycle would be $p_i + c_{i,x}$. Thus the optimal tour is $\forall_i \min(p_i + c_{i,x})$.

p_i is calculated through the creation of multiple scoring grids, S, which are created and filled through a recursive relationship. 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:

$$S_{X,i} = \begin{cases} c_{1,i} & |X| = 2 \\ \min\left(S_{X\setminus i,j} + c_{ji}\right) & j \in X \wedge j \neq i \wedge j \neq 1 \end{cases} \quad (6)$$

Therefore for a set of size n, $n-2$ subsets are considered, each of size $n-1$ excluding node n.

III. Accelerating Optimisation Problems By CUDA

A. Outline of CUDA

CUDA, the Compute Unified Device Architecture, is a programming language developed by NVIDIA, to enable the implementation of GPGPU programs on NVIDIA hardware. Briefly introduced is the computational model adapted by CUDA to provide a background for concepts discussed in this paper, but for full details readers are referred to the papers [4], [5].

CUDA adopts a Single-Instruction; Multiple-Thread (SIMT) programming model. Threads are organised into 3D groups called *blocks*, and groups of blocks are organised into larger 3D groups called *grids*. The dimensions of the blocks and grids only serve as an abstract representation to aid with indexing data.

Blocks can be executed in any order, therefore they must be independent of each other. Once computation has begun on a block, it will execute to completion before unloading again. Threads are dispatched to be executed in a group called a *warp*. All threads in a warp follow the same execution path; therefore, code divergence in CUDA must be minimised. When divergence is encountered in CUDA, the warp is partitioned such that all threads that satisfy one code path are executed, and threads that do not satisfy that path remain idle. When execution of the path completes, the warp is restarted and the other path(s) executed.

Generally, the memory employed by CUDA is slow. *Global* memory is accessible across all blocks, but has relatively high access times, as do the other global stores of *texture* and *constant*. *Shared* memory is quicker and is accessible by all members of the same block; additionally, registers are available on a per thread basis. In an effort to hide the slow global memory access times, CUDA will context switch other warps in whilst waiting for memory transactions to complete, in order to maximise *transactions in flight*.

B. Applicability of GPGPUs to Optimisation Problems

GPGPU implementations of solving methodologies for optimisation problems are often highly problem specific, or merely GPU re-implementations of established heuristic solving methodologies.

In the context of DP, GPUs can be considered to be suitable as the matrix based structures at the core of DP match the architecture of CUDA very well. However, it is challenging to implement and parallelize a DP effectively due to the large memory complexity, a problem which is compounded by the relatively slow memory speeds offered by the GPU. Furthermore, dependencies between blocks can cause problems, as blocks in CUDA can be executed in any order. These problems are addressed by the model proposed in this paper.

C. Existing Research

We now consider the existing methodologies of parallelising these problems and algorithms available in the literature, in order to identify shortcomings in existing methods. Some existing concepts are incorporated into our model.

Algebraic DP, a high level abstract branch of DP, has been parallelised effectively using the GPU [6], by iterating through the scoring grid in a series of *diagonals*. This is because once a diagonal of the grid is calculated, the dependencies for the next have been satisfied. This is effective for algebraic DP; as can be observed, the scoring grid and dependencies will always take the same structure.

We also consider methods tailored to the selected test problems individually, rather than looking for a generic solving method. In the case of the LCS Problem, again the approach of iterating through the grid in a series of diagonals is adopted [7]. This can be enhanced further using some problem specific techniques such as decomposing the grid to better match the underlying architecture of the GPU [8] or pre-computing some additional dependencies first, to allow for greater parallelism [9].

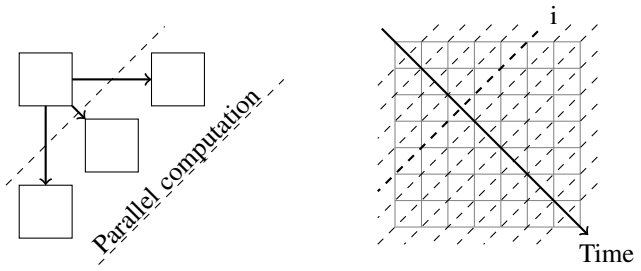
In the case of the knapsack problem, there is only one example of GPU DP in the literature, and this demonstrates that the scoring grid can be restructured such that each row of the matrix can be solved in parallel [10]. Otherwise there are standard heuristic methods available such as evolutionary algorithms [11] and branch and bound [12].

The travelling salesman problem has no exact solving methodologies on the GPU that we are aware of, and this is most likely due to the memory constraints posed by GPU programming. There are however a variety of heuristic solving methods such as genetic algorithms [13] and iterative solution improvement [14].

IV. Proposed Model

A. Model Design

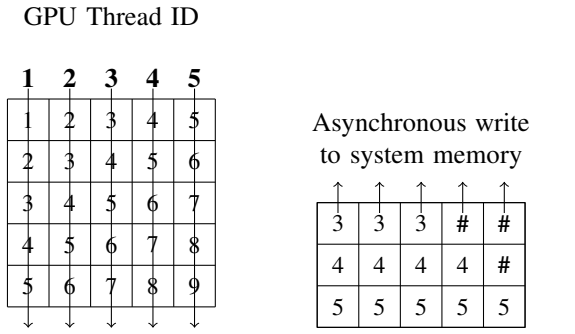
The model that we propose aims to be a semi abstract model. It is demonstrated on the three aforementioned problems, but we assert that it can be applied to others with the minimum of adaptation. In some cases our method may not be as quick as



(a) Once a cell is filled, the dependency for many others are satisfied

(b) Iterating through the scoring grid in a *diagonal* fashion

Fig. 1. A graphical representation of the *'diagonal'* parallel method adopted



(a) The GPU threads calculating the iterations in parallel

(b) The linear memory structure and preallocated memory addresses

Fig. 2. The thread and memory structure adopted. Numbers within the cells represent the iteration that is currently being calculated

state-of-the-art for every given problem; this is a side-effect of aiming to stay generic.

It can be observed that in the DP definition of all the introduced problems, once one *step* of the problem has been solved, the dependencies for several other steps have been satisfied, shown in Figure 1a. As shown, iterating through the grid in an diagonal based fashion is an established method for DP on the GPU [6], [7], and is demonstrated in Figure 1b. Whilst this approach has been detailed in the literature for specific problems, most notably the LCS problem, to the best of our knowledge it has never before been approached in a generic manner.

Managing memory on the GPU using CUDA is a challenge due to the restrictive amount as well as low access speeds. Because of this, careful consideration must be made to minimise memory requests, and accelerate remaining memory transactions. In CUDA, should memory accesses take place from sequential addresses, CUDA will *coalesce* the memory requests. This is a process by which memory can be read or written, in a single transaction, by every thread in a warp simultaneously.

Therefore, as well as restructuring the DP grid such that all memory is linear, divergence in the code should be reduced as far as possible. This is due to the fact that if even one thread requires a different code path to the others in the warp, its memory access will not be coalesced and an entire memory transaction will be wasted. This is achieved through pre-allocating additional memory so that all threads within a warp have a location to read and write to, albeit redundantly, so as not to cause divergence. The numbers within the grids in Figure 2a represent the iteration of the algorithm, and Figure 2b demonstrates the memory structure in use.

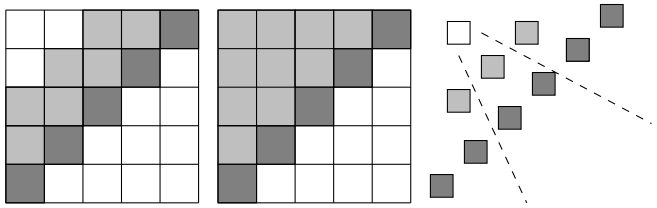
One of the biggest limitations of programming with CUDA is the slow transfer speed between the *host*, the CPU and system memory, and the *device*, the GPU. We overcome this by using asynchronous memory transfer, a feature available in all modern CUDA cards. This allows data to be transferred back to the host whilst the device continues computation. Managing the memory this way also means that it alleviates the limitation of memory complexity associated with DP. The GPU does not have to maintain the whole DP grid, only cells that are required by immediate dependencies; the rest can be asynchronously pushed back to the much larger system memory with no interruption in computation. In the case of problems that have a very small number of cell dependencies, such as the LCS, this allows much larger problems to be solved through CUDA than was previously possible.

In Figure 2a the thread pattern adopted is depicted, and how the diagonal iterations through the grid are mapped to the multiple threads. As already discussed, whilst some threads will be idle at the beginning and end of execution, periods known as *warm-up* and *warm-down*, they will still be following the same code path to prevent any divergence.

B. Code Enhancements

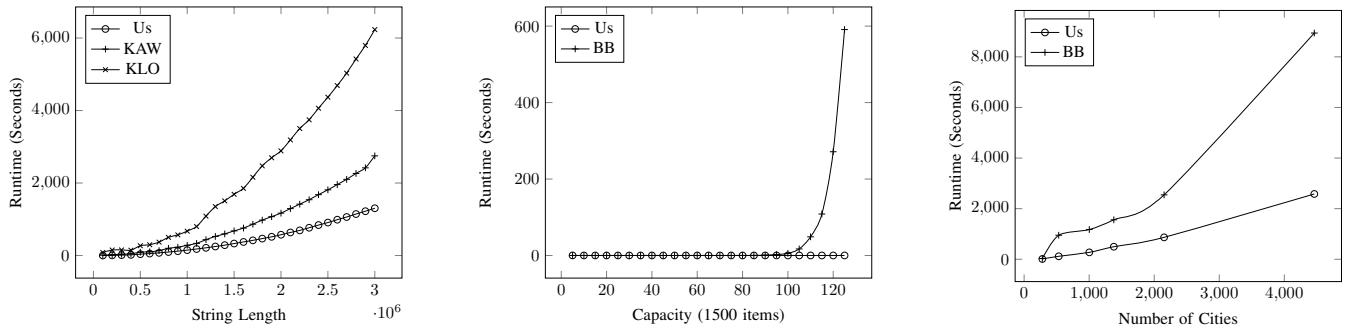
Some optimisations can be applied to the model so that it can be accelerated further to cope with the vast computational complexity of the problems. As well as the memory model already detailed, we take further steps to optimise the memory structure further based on [15].

All fixed data is stored in the texture memory of the GPU, for example the test data in the LCS and KS problem, or the travel time matrix in the TSP. This is due to the optimisations available for spatially close data, and most frequently test data is requested sequentially (LCS) or spatially (TSP). This also allows multiple elements to be packed into one data type, minimising memory transactions.



(a) LCS (b) KS (c) TSP

Fig. 3. Representations of how the problems are adapted to fit the model. Dark grey cells denote the current iteration, and light grey are dependencies that must be maintained



(a) LCS (b) KS (c) TSP

Fig. 4. The runtime of the proposed model on the introduced problems, across varying test instance sizes

C. Adapting the Model

Detailed here is how the mode is adapted to suit the specific problems. In all instances test data is stored in texture memory.

1) Longest Common Subsequence: The LCS problem is dependent on the previous two iterations; this allows any other completed iterations before that to be pushed to system memory, see Figure 3a. Due to this it has a low memory complexity on the GPU.

2) 0-1 Knapsack Problem: The knapsack problem is dependent on the previous $j - W$ iterations. This means that the whole scoring grid must be maintained until the halfway point. From that point the oldest iterations can be pushed to system memory; see Figure 3b.

3) Travelling Salesman Problem: The travelling salesman problem requires considerably more adaptation. Due to the recursion, each iteration is considerably larger than the last, meaning that a square grid is inappropriate. Therefore at each step, the recursive steps are calculated in parallel, but each iteration requires a new vector scoring grid to store the current diagonal, see Figure 3c.

V. Evaluation

A. Testing Configuration

All GPU testing will be carried out using an NVIDIA Titan GPU, which contains 2688 CUDA cores clocked at 837Mhz and 6GB of memory. The CPU in the system is an Intel 3930k at 3.2GHz with 16GB of system memory running Arch Linux. Average runtimes will be generated across five replicate runs of each test. The problems were compiled using CUDA 6 and GCC 4.9.

Where possible we benchmark the proposed model against the equivalent CPU or GPU DP implementations, depending on what is available in the literature. Due to space constraints, these will not be detailed here, but the reader will be referred to the relevant literatures.

1) The Longest Common Subsequence Problem: The longest common subsequence problem will use randomly generated test data, where the length of the match sequence between the two sequences is restricted to 50%. Our implementation of the LCS problem will be compared against the GPU algorithms proposed by Kawanami [16] (KAW) and Kloetzli [8].

2) The 0-1 Knapsack Problem: In the case of the KS problem the capacity of the knapsack has a large impact on the complexity, as well as the number of items. Due to this increasing both the capacity and the number of items simultaneously will have a large impact on runtime at each step. Therefore the number of items is fixed at 1,500, with the capacity being incremented between tests. Values and weights will be assigned at random, whilst ensuring all values are less than 30% of the total capacity. This is compared against a branch and bound approach (BB) [17].

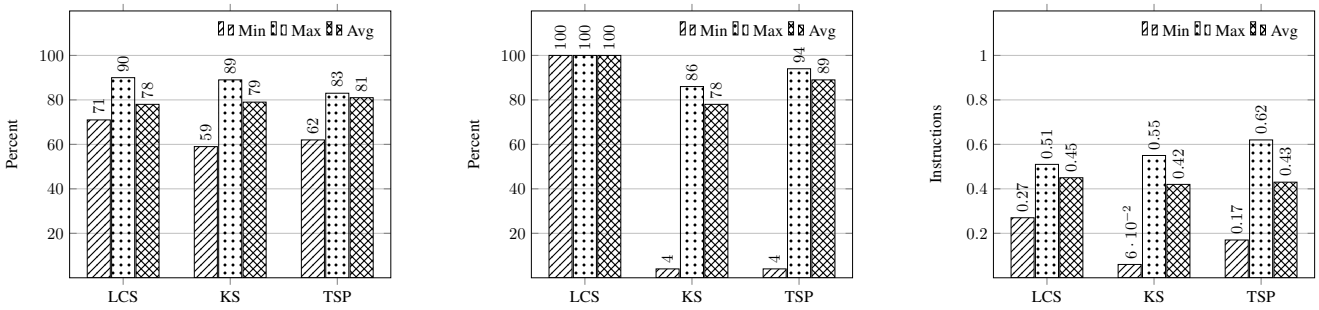
3) The Travelling Salesman Problem: Test instances for the TSP problem are retrieved from TSPLIB [1]. The instances we have selected are a280, att532, pr1002, nrw1379, u2152, fnl4461, with the number in the problem name denoting the amount of cities considered in the problem. We have selected these instances as they provide a range of problem sizes. There are very few exact approaches to the TSP problem available for the GPU available in the literature - nearly all are inexact heuristic methods - therefore we compare against a branch and bound method implementation [18].

B. Computational Results

Due to the nature of exact solving methodologies, as already outlined in this paper, CPU implementations have large running times. This causes a large differential in the timings between the CPU and the GPU. Also as there are very few, or in some instances not any, exact implementations available on the GPU, finding reference implementations for comparison is challenging. Therefore simple runtime execution profiling is not the most effective method of evaluating the proposed model, but it does provide a basic insight to performance and runtimes. Efficiency of the model is more carefully considered in the following section.

In the case of the LCS problem, see Figure 4a, our model is the fastest implementation, in all tests. Kloetzi provides the slowest implementation, and it is our belief that the fluctuations in runtime are caused by certain problem sizes not mapping exactly to the hardware or algorithmic structure, as these fluctuations persist across replicate runs. Kawanami is quicker, as it is in essence an augmented version of Kloetzi.

[1] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/



(a) Global load efficiency (b) Warp execution efficiency (c) Instructions per clock

Fig. 5. The results of the introduced different CUDA profiling metrics

Considering the KS problem, see Figure 4b, again our model demonstrates the highest level of performance, but it is only being compared against a CPU implementation. What the results demonstrate is that when using a CPU, the runtime of exact solving quickly rises to unmanageable levels as the capacity increases, although this is an issue the GPU does not suffer from. The GPU runtime however is not linear, the runtime does increase, although at a much slower rate than the CPU. This makes a direct comparison impossible as by time the GPU runtime rises, the CPU is taking many days to run.

Finally considering the TSP problem, see Figure 4c, our model shows considerably improved performance against the CPU. It was challenging to find a reference implementation to compare our model to, as the most prevalent method of solving the TSP by far is through the use of heuristic and metaheuristic methods. However, we sought an exact method to compare ours too.

In all of the selected problems, our model demonstrates improved run times compared to existing methods. In some of the test instances we used CPU methods to compare against in order to ensure that our exact model was being compared against similar algorithms. In the case of the LCS problem, comparisons were made against GPU implementations, as there are many readily available in the literature, and the proposed model still provided the quickest run times. As already discussed we expected to suffer some speed disadvantages as we are proposing a generic multi-problem approach, although the run times observed demonstrate a high level of performance from the model.

C. CUDA Profiling

In addition to considering the runtime of each algorithm, the efficiency of the model can be considered using the NVIDIA CUDA profiler, *nvprof*. The metrics that we have selected are global load efficiency, warp execution efficiency and the instructions per clock.

The global load efficiency is a measure of the requested global memory load throughput compared to the required global memory load throughput. Using this metric we can get an indication of how the memory structures align to the underlying hardware, and how effective the implemented memory access patters are. Secondly the warp execution efficiency is a ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor. Using this metric we can gauge how much divergence is within the model and how effectively the threads available on the GPU are being used. Finally recording instructions per clock, which should be maximised, is an effective overall measure of performance, as many factors can have an effect on it such as memory transactions and divergence within the code.

Due to the nature of the profiling, there is considerable overhead added to the runtime. Each CUDA code block has to be replicated multiple times, and profiling data saved and stored. Problem sizes had to be severely restricted therefore in the case of the LCS problem string length was 3,000, in the case of the KS problem capacity was 500, and in the case of the TSP problem the smallest a280 instance was used.

Considering the global load metric first, the results show that all averages are greater than 75%. This demonstrates that the underlying memory architecture is working efficiently, and even the values for the slowest accesses are still 60% efficient or higher. As memory operations on CUDA are computationally expensive, and these dynamic programming based approaches require large amounts of memory operations it is imperative the efficiency value is high, which based on the observed results, it is.

Next considering the warp execution efficiency, all averages in this test are nearly 80% or higher. The LCS problem shows 100% across all runs; this is due to padding being placed at the start and the end of the string allowing it to be exactly divisible by the block size, removing the need for any code divergence at all. The other problems have very low minimum values which can likely be attributed to very small diagonals at the start and the end of execution, during the *warm up* and *warm down*, wasting resources within a block and causing divergence. Regardless, the averages are still high, further demonstrating the low minimums are isolated incidents. The results demonstrate the resources of the GPU are being utilized effectively during execution.

Finally considering the instructions per clock (IPC), all of the averages are 0.5 or greater. Whilst this value is not as high as the other two metrics, it should be remembered that the IPC will be lower in memory heavy algorithms such as this one as instructions will stall whilst waiting for memory operations.

Again the low minimums can be attributed to fringe cases, as the averages still remain high.

VI. Conclusions and Future Work

In this paper we have proposed a generic GPU based model that allows the solving of optimisation problems, exactly, through dynamic programming based methods. The model we have presented is applicable to a range of problems with a small amount of adaptation for each, including others that have not been introduced here. We have also developed and outlined a simple, yet effective memory model that allows large scale problems to be solved using the GPU. This memory model is applicable to a different optimisation problems, that can be solved using a grid based DP approach. The only small adaptation required for each problem is the amount of already calculated rows to maintain in memory, before asynchronously pushing them to system memory.

The presented model has demonstrated that it performs quicker than existing exact solving methodologies available on both the CPU and on the GPU. Metrics have validated that the model uses the resources presented by the GPU efficiently. Furthermore, it now allows problems that were previously too large to be considered by exact solving methodologies, to be solved exactly rather than relying on inexact methods such as meta-heuristics. Thus we believe the objective of the model and this paper have been achieved.

Future enhancements would consider how the divergence within the code could be handled more effectively which would raise the warp execution efficiency, and likely also the IPC. An alternate route for further development would be multiple GPU implementations which should allow significant performance gains with only minimal code alterations. It must be remembered that the core objective of this model is to be problem agnostic as far as possible, so any enhancements would have to be universally applicable as far as possible.

References

[1] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimisation*, 2nd ed. Wiley-Interscience, July 1999.

[2] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, 4th ed. Athena Scientific, 2007, vol. 2.

[3] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer Programming Formulation of Traveling Salesman Problems," *Journal of the ACM*, vol. 7, no. 4, pp. 326–329, Oct. 1960.

[4] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, July 2010.

[5] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed. Morgan Kaufmann, Dec 2012.

[6] P. Steffen, R. Giegerich, and M. Giraud, "GPU Parallelization of Algebraic Dynamic Programming," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin Heidelberg, 2010, vol. 6068, pp. 290–299.

[7] A. Dhraief, R. Issaoui, and A. Belghith, "Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability," in *Proceedings of The First International Conference on Advanced Communications and Computation (INFOCOMP)*, 2011, pp. 143–148.

[8] J. Kloetzli, B. Strege, J. Decker, and M. Olano, "Parallel Longest Common Subsequence Using Graphics Hardware," in *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, ser. EG PGV'08. Aire-la-Ville, Switzerland: Eurographics Association, 2008, pp. 57–64.

[9] J. Yang, Y. Xu, and Y. Shang, "An efficient parallel algorithm for longest common subsequence problem on GPUs," in *Proceedings of the World congress on Engineering (WCE)*, vol. 1, 2010.

[10] V. Boyer, D. E. Baz, and M. Elkihel, "Solving knapsack problems on GPU," *Computers & Operations Research*, vol. 39, no. 1, pp. 42 – 47, 2012, Special Issue on Knapsack Problems and Applications.

[11] P. Pospíchal, J. Schwarz, and J. Jaros, "Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU," in *Proceedings of the 16th International Conference on Soft Computing (MENDEL)*, 2010, pp. 64–70.

[12] M. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, May 2012, pp. 1769–1777.

[13] S. Tsutsui and N. Fujimoto, "Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, ser. GECCO '09. New York, NY, USA: ACM, 2009, pp. 2523–2530.

[14] N. Fujimoto and S. Tsutsui, "A Highly-Parallel TSP solver for a GPU Computing Platform," in *Numerical Methods and Applications*, ser. Lecture Notes in Computer Science, I. Dimov, S. Dimova, and N. Kolkovska, Eds. Springer Berlin Heidelberg, 2011, vol. 6046, pp. 264–271.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "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*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82.

[16] K. Kawanami and N. Fujimoto, "Facing the multicore-challenge ii," R. Keller, D. Kramer, and J.-P. Weiss, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. GPU Accelerated Computation of the Longest Common Subsequence, pp. 84–95.

[17] S. Martello, D. Pisinger, and P. Toth, "Dynamic programming and strong bounds for the 0-1 knapsack problem," *Management Science*, vol. 45, no. 3, pp. 414–424, 1999.

[18] M. Held and R. Karp, "The traveling-salesman problem and minimum spanning trees: Part ii," *Mathematical Programming*, vol. 1, no. 1, pp. 6–25, 1971.