A Parallel Framework for Horizontally Local Dynamic Programming Problems

Rajesh Kumar*, Kishore Kothapalli†
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500 032
Email: *rajesh.kumar@research.iiit.ac.in, †kkishore@iiit.ac.in

***Abstract*—The technique of Dynamic Programming (DP) is used to solve a wide variety of combinatorial search and optimization problems. For a subset of these problems, each update to a cell in the DP table is a function of the contents of the previously computed cells belonging to the immediately previous row. We call these problems *Horizontally Local Dynamic Programming (HLDP)* problems.**

In this paper, we develop a parallel GPU based framework for solving `HLDP` problems. We categorize them depending upon the distribution and position of dependencies. We propose suitable techniques for each of these categories. Finally, we consider several case study problems to show how our framework can be used.

I. Introduction

Dynamic Programming (DP) is an effective and robust technique that is used in solving a variety of optimization problems from domains such as scheduling, string editing, packaging, bioinformatics, and inventory management [7]. The importance of dynamic programming to parallel computing can be seen from the fact that dynamic programming is included as one of the thirteen dwarfs identified by a seminal Berkeley report [1]. Dynamic programming involves solving a problem by dividing it into smaller subproblems. Problems that can be solved using dynamic programming present two essential properties: *optimal substructure* and *overlapping subproblems*.

The *optimal substructure* property indicates that that the solution to a problem can be obtained by blending the optimal solutions to its subproblems. The *overlapping subproblems* property indicates that the number of unique subproblems is usually smaller. In other words, the subproblems resulting from decomposing a problem repeat and hence their solutions can be computed only once and reused other times.

Dynamic programming solutions take one of the two approaches: *top-down* or *bottom-up*. In the top-down approach, the solution is formulated recursively using the solution to its subproblems. Since the subproblems overlap, one can use memoization to record the solutions of previously computed subproblems into a table for later use. In the bottom-up style, all the subproblems of a the smallest size are solved and these solutions are used to create a solution to bigger subproblems. Usually, one uses an iterative approach in constructing solutions to bigger subproblems using solutions to smaller subproblems.

`HLDP` (Horizontally Local DP), a special class of DP problems, represents problems in which the contents of the cells from the immediately previous row/column decide the update to an entry in the DP table. Examples of this class of problems are the Non-Segmental Dynamic Time Warping problem [11], the knapsack problem [2], the checkerboard problem [9], and the subset sum problem [8]. Many other DP problems can also be modeled as `HLDP` problems.

For every `HLDP` problem, one can associate a table whose values can obtained iteratively using the bottom-up style of solving dynamic programming problems. In general, one can visualize a formula that governs how the entries in the table are filled. The general form of the `HLDP` formula can be given as below where the function f depends on the problem.

$$table[i][j] = f\left(\; table[i-1][j+x] \mid x \in g(i,j) \; \right)$$

Here $g(i, j)$ is a function which returns a set of integers.

In this paper, we start by categorizing `HLDP` problems into five broad classes. We design strategies for each of these five categories of problems as a complete parallel framework. We envisage that our framework can help end users improve their productivity while programming current heterogeneous computing platforms. Using our framework, one can readily create parallel programs for problems that follow the `HLDP` approach by specifying and programming the function that guides how the dynamic programming table is filled. The nature of the function also indicates the dependency pattern.

Since the dependency distribution varies across the five categories, we present independent execution models for each class. Our focus here is to exploit the inherent advantages and counter the inherent disadvantages that arise due to the dependency distribution.

A brief summary of noteworthy technical contributions of this paper are lised below.

- A categorization of `HLDP` problems into five categories based on the dependency distribution is obtained.
- A parallel framework with high level support for each category of problems is developed. A user of the framework needs to supply only the problem dependent function f.
- Several optimizations such as tiling to exploit locality, explicit caching and coalescing to reduce memory access latencies, and heterogeneity wherever possible to maximize the use of available resources are introduced in the framework.

978-1-4673-8286-1/16 $31.00 © 2016 IEEE
DOI 10.1109/IACC.2016.49

- Four case studies that illustrate the usage of our framework are presented.

We note that there might exist problem specific optimizations that help in improving the performance of parallel algorithms on specific HLDP problems. In this work, our interest is however on problem independent optimizations that can improve the performance of parallel algorithms for the entire class of HLDP problems.

A. Related Work

Dynamic programming has direct applications in different domains. It is thus not surprising that a lot of research attention is devoted in parallelizing DP problems. While problem specific solutions are best way to gain performance, generic solutions/frameworks enable the programmers and domain experts to extract reasonable performance with little effort. Chowdhury et al. [3] presents algorithms based on the class of the DP problems (Local Dependency problems, Gaussian Elimination Paradigm problems, Parenthesis problems). Chowdhury et al. [4] also developed cache-oblivious algorithms for dynamic programming problems in bioinformatics. Kumar et al. [9] presented a parallel heterogeneous framework for Local Dependency DP (LDDP) problems which reduces the effort required by the programmer to write effective parallel solution for this class of problems.

Problem specific solutions for some of the problems belonging to HLDP class have reached from primitive parallel solutions to advanced level. Knapsack problems have seen great advancement from early parallel solutions ([10] and [6]) to very fast GPU based parallel solutions([2]). Like other DP problems, techniques like dominance technique ([5]) has been exploited well for knapsack problems.

B. Organization of the paper

The rest of this paper is organized as follows. Section 2 details the required prior material. In Section 3, we describe the heterogeneous computing platform used in our work. Optimization of framework is described in Section 4 followed by the implementation details in Section 5. Four case studies that use our framework are described in Section 6. The paper concludes in Section 7.

II. Preliminaries

The class of *HLDP* problems is characterized by a function which considers values of previously computed cells from the immediate previous row to update each position in a 2 dimensional table. The value to be filled in $cell_{i,j}$, the ith row and the jth column, of the table is defined by following function.

$$cell_{i,j} = f\left(\ cell_{i-1,j+x} \mid x \in g(i,j) \ \right)$$

Here, the function $g(i,j)$ returns a set of integers such that $j + g(i,j)$ indicates which cell(s) from the previous row shall be considered for computation.(See Figure 1(a)). Function f indicates how the value to be filled in $cell_{i,j}$ will be computed using those cells.

The set returned by $g(i,j)$ determines the nature and distribution of dependency across different cells. For a given row i, if $g(i,j)$ returns same set for all columns, we call the problem *Uniform HLDP problem*. (See Figure 1(b)). Else we call it *non-uniform HLDP problem*.(See Figure 1(c)).

The most straightforward way to parallelize any HLDP problem is to process the table row by row and operate on all columns of the row under consideration in parallel. The degree of parallelism remains constant with time.(See Figure 1(d)).

A. A Brief Overview of our Experimental Platform

In our experiments, we use an NVidia Tesla K20 GPU and an Intel i7 980x CPU. We use CUDA API Version 5.0 for programming the GPU and use the OpenMP Specification 3.0 for programming the CPU. The Tesla K20 GPU comes with 13 streaming multi-processors (SMX) each housing 192 cores for a total of 2496 cores. Each core is clocked at 706 MHz, and has an L2 cache of 1.25 MB. The K20 GPU can provide a peak throughput of 3.52 TFLOPS of single-, and 1.17 TFLOPS of double precision floating poing operations.

The Intel i7 980 is a six-core machine with each core running at 3.5 GHz. With active SMT (hyper threading), each core can support two logical threads. Each core has a 32 KB instruction and a 32 KB data L1 cache, and a 256 KB L2 cache. All the six cores share an L3 cache of 12 MB.

III. Description of our Framework

In this section we first explain the categorization we propose for *HLDP problems*. Recall that our categorization is primarily based on the nature of distribution (uniform/non-uniform) of dependencies. Uniformity in dependency distribution offers further scope for sub-classification. These sub-classes have some unique properties which form the basis of the design of our framework. We start with uniform dependency HLDP problems (Section III-A) and proceed to non-uniform dependency HLDP problems in Section III-B.

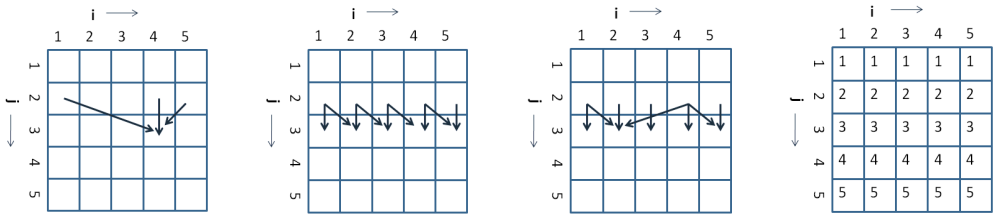
A. Uniform HLDP Problems

Recall that the criterion of uniformity states that the function $g(i,j)$ returns the same set for all the cells in a given row. In addition to satisfying the criteria of uniformity, we introduce two additional criteria in the following that allows us categorize the uniform HLDP problems further.

1) *Contiguity*: The function $g(i,j)$ must return a set having consecutive integers.
2) *1-Sidedness*: The function $g(i,j)$ must return a set of either non-positive integers or non-negative integers.

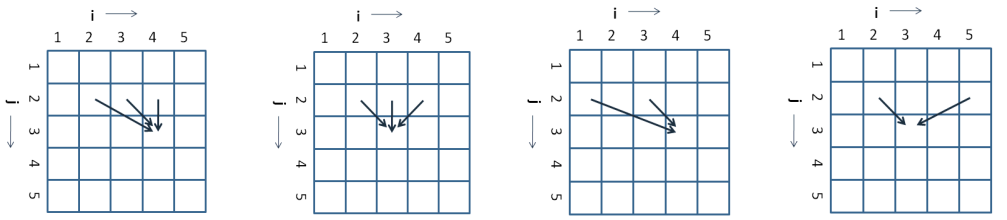
Depending on the which subset of the above two criteria are met by the problem, we will now have four categories of uniform HLDP problems as discussed in the following.

1) 1-Sided, Contiguous: Problems belonging to this subclass satisfy the criterion of contiguity as well as the criterion of 1-sidedness (See Figure 2(a)). For these problems, we decompose the table into many sub-matrices (called blocks). We operate on these blocks in a fashion that maintains the data dependency. In comparison to operations performed on entire



(a) Explanation of $g(i, j)$ (b) Uniform Function (c) Non-uniform Function (d) Nature of parallelism

Fig. 1. (a) Dependencies for $cell_{3,4}$, if the function $g(3, 4)$ returns the set {-3, 0, 1}. (b)All the cells in row 3 have similar (uniform) access pattern. The function g returns the set {-1, 0}for all the cells of this row. (c) All the cells in row 3 have dissimilar (non-uniform) access pattern. The function g returns different sets for different cells of this row. (d) All the cells marked with number i can be processed in i^{th} iteration. Cells marked with same the number can be processed in parallel.



(a) 1-Sided Contiguous (b) 2-Sided Contiguous (c) 1-Sided Non-contiguous (d) 2-Sided Non-contiguous

Fig. 2. (a) $g(3, 4)$ returns the set {-2, -1, 0}. All elements are consecutive and non-positive integers. (b) $g(3, 3)$ returns the set {-1, 0, 1}. All elements are consecutive but it contains both negative and positive integers. (c) $g(3, 4)$ returns the set {-1, -3}. All elements are non-positive but they are not consecutive. (d) $g(3, 3)$ returns the set {-1, 2}. It contains non consecutive integers (both negative and positive).

row, operating on blocks results in improved data locality. This leads to better data reuse as the blocks can be loaded into faster levels of memory. We assign each block to a block of threads (on GPU). Each block of threads is responsible for loading the appropriate data in shared memory of the SM on which it is scheduled. This causes in reduction of reads/writes required from/to global memory.

We intend to maximize parallelism at both levels (between the blocks and within the blocks). The obvious choice for this seems to be following the same pattern (the horizontal pattern) for operating on inter-block level as well as intra-block level. But this violates the dependency. The next least restrictive choice is to follow the anti-diagonal pattern at inter-block level as shown in Figure 3 and horizontal pattern at intra-block level. This prevents dependency violation for problems exhibiting 1-sided dependency. So in this case (1-Sided,Contiguous), we use blocking. Apart from loading the main block, we also need to load one additional dependency window to the shared memory. The dimension of this window is same as dimension of the block/tile. We call it *single window tiling* and is illustrated in Figure 4.

2) 2-Sided, Contiguous: Problems belonging to this subclass satisfy the criterion of contiguity but not the criterion of 1-sidedness (See Figure 2(b)). Because of the 2-sided dependency, the tiling approach explained in III-A1 does not work here. We use the modified version of heterogeneous approach proposed by Kumar et al. [9] for solving LDDP problems following horizontal pattern. In a pure GPU (or CPU) approach, we can assign one thread (or block of threads)

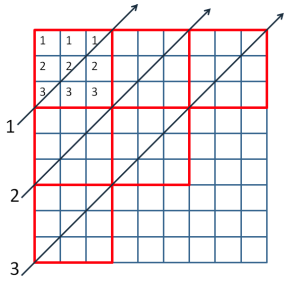


Fig. 3. Blocking Strategy for 1-sided, Contiguous problems. Blocks are processed in an anti-diagonal fashion. Blocks lying across i^{th} anti-diagonal can be processed in i^{th} iteration. Within the block, we can process the j^{th} row of the tile in parallel, after finishing $(j-1)^{th}$ row of the tile

to each cell (or a set of cells) of a row. Since enough amount of work is available in all the iterations, we can distribute the workload between the CPU and the GPU from the first iteration.

The modified approach is as follows. For $numberOfRows$ iterations, distribute the work between the CPU and the GPU. Assign first t_{share} cells of the corresponding row to the CPU and rest to the GPU. Unlike discussed in [9], we might need to transfer more than one cell depending on the problem. Usually this does not affect the performance since all these dependent cells are scheduled to be transferred together. We discuss the calculation of the parameter t_{share} in section V.

3) 1-Sided, Non-contiguous: Problems belonging to this subclass satisfy the criterion of 1-sidedness but not the cri-

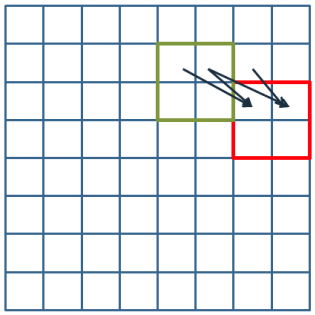


Fig. 4. Single Window Tiling. If g(i,j) returns the set {-1, -2}, to compute the red block starting at row number 3 and column number 7, we need to load the green window, starting at row number 2 and column number 5 into the shared memory. In the case of 1-Sided contiguous problems, only one such window needs to be copied.

terion of contiguity (See Figure 2(c)). Subsection III-A1 explains the blocking strategy and its applicability for problems exhibiting 1-Sided dependency. Same blocking strategy works here with some modifications to take care of discontinuity of dependencies. In this case, we need to load multiple additional dependency windows (problem dependent) to the shared memory. We call it *multi-window tiling* and is illustrated in Figure 5. The dimension of all these dependency windows is similar to the dimension of the block/tile.

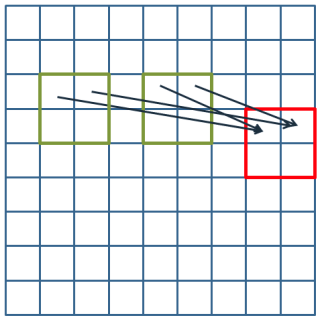


Fig. 5. Multi Window Tiling. If $g(i,j)$ returns the set $\{-3,-6\}$, to compute the red block, we need to load the both the green windows, starting at row 3, column 2, and starting at row 3 and column 6, into the shared memory. In the case of 1-Sided non-contiguous problems, the number of such windows are the determined by the length of set returned by function $g(i,j)$.

4) 2-Sided, Non-contiguous: Problems belonging to this subclass neither satisfy the criterion of contiguity nor the criterion of 1-sidedness (See Figure 2(d)). Similar to that of 2-Sided Contiguous problems, these problems can also be approached using heterogeneous approach explained in section III-A2. The only difference is that we must take care of discontinuity in the distribution of data. For this we batch together the discontinuous chunks into one larger chunk and send it together.

B. Non-uniform HLDP Problems

If the function $g(i,j)$ does not return the same set for all the cells of a given row, we call them *Non-uniform HLDP Problems*. Because the set returned by $g(i,j)$ may vary in length and content, it is difficult to exploit any pattern in the dependencies.

One important thing to note here is that, in non-uniform `HLDP` problems, the number of memory accesses needed is not the same for all cells. This property is an indicator for judicious use of memory hierarchy. The key to performance is to load the faster levels of memory (shared memory in this case) with most frequently used cells. However, sorting the cells (with number of memory accesses as parameter) of a given row is a costly operation. We use *explicit caching* for solving this problem. We create a temporary array in the shared memory. Every cell of the row under consideration (of the actual table) is mapped to one cell in the temporary array. Each cell in the temporary array stores the tuple (tag, value), where tag points to the actual cell (among all eligible cells) present in the shared memory.

IV. Optimizations

In the following subsections, we describe some optimization techniques used in our framework. These optimizations are applicable independent of the actual problem.

A. Thread per Cell vs Thread per Block

Current generation CPUs do not offer good performance when one creates threads in large numbers for reasons such as thread creation and context switching overheads. It is therefore practical to create a small number of heavy-weight threads on the CPU. Each CPU thread is responsible for processing a contiguous portion of the table. The GPU, on the other hand, is amenable for creating a large number of light-weight threads and use its massive parallelism.

B. Memory Coalescing on the GPU

One of the prime factors affecting the performance of GPU algorithms is their use of memory coalescing. To improve memory coalescing, we store the table in row order fashion so that threads can share data read in chunks by other threads.

C. Managing Data Transfers

Data transfer between the CPU and the GPU is required when one uses both the CPU and the GPU in an heterogeneous manner. Since, data has to be transferred both ways (CPU to GPU and GPU to CPU), we can not overlap data transfer with computation. However, we overlap both the data transfers using different streams. Since we only transfer a small amount of data corresponding to cells that are enough to satisfy the dependency constrains, we make use of the pinned memory feature that provides fast memory access for small data sizes.

V. Implementation Details

A. Determining Tile Dimensions

We determine tile dimensions empirically by running the algorithm for different tile sizes and tracing the size which leads to minimum running time. As shown in Figure 6, independent of the table size, the optimal tile size remains the same.

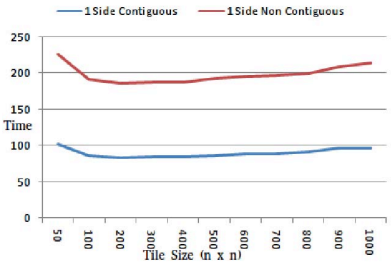


Fig. 6. Running time (in milliseconds) for different tile sizes (n x n) for non-segmental DTW problem (1 Side Contiguous) and Knapsack Problem (1 Side Non-contiguous) of table size 16k x 16k

B. Finding the value of t_{share}

The number of cells processed by the CPU per iteration is defined by t_{share}. We find this empirically by running the algorithm on a sample row (for one iteration). We compare the time taken by the host and the device and distribute the work accordingly.

C. Using the Framework

Using this framework is fairly straight-forward. A user has to provide a program in CUDA for the function f defining how the $cell_{i,j}$ of the table will be filled using the values of the neighboring cells. The nature of this function can be used to indicate the category of the `HLDP` problem. In addition, the user has to specify the initial values of the cells in the table.

VI. Case Studies

We present the performance of our framework on four different problems. Three of the problems cover the four cases of uniform `HLDP`, and one covers the non-uniform `HLDP` case. Since a similar approach is used in our framework for 2-Sided Contiguous and 2-Sided Non-Contiguous Problems, in principle, there are only three implementation categories for uniform `HLDP` problems.

A. Non Segmental Dynamic Time Warping (NS-DTW)

NS-DTW is used for Query-by-Example Spoken Term Detection (QbE-STD) [11]. In this problem, we have a spoken query Q containing n feature vectors, and the spoken reference R containing m feature vectors. The distance between the i^{th} feature vector of Q and the j^{th} feature vector of R is denoted as $d(i,j)$. NS-DTW uses only the local constraints to obtain the region in the reference R that is likely to contain Q. The method includes construction of a similarity matrix S and a transition matrix T. Both matrices are of size $(m \times n)$ and their cells are filled according to following equations.

$$cS(i,j) = min \begin{cases} \frac{d(i,j)+S(i-1,j-2)}{S(i-1,j-2)+w_e} \\ \frac{d(i,j)+S(i-1,j-1)}{S(i-1,j-1)+w_d} \\ \frac{d(i,j)+S(i-1,j)}{S(i-1,j)+w_s} \end{cases}$$

$$T(i,j) = \begin{cases} T(i-1,\hat{j}) + w_e & \text{if } \hat{j} = j-2 \\ T(i-1,\hat{j}) + w_e & \text{if } \hat{j} = j-1 \\ T(i-1,\hat{j}) + w_e & \text{if } \hat{j} = j \end{cases}$$

where,

$$\hat{j} = \underset{\hat{j} \in \{j,j-1,j-2\}}{\text{argmin}} \begin{cases} \frac{d(i,j)+S(i-1,j-2)}{S(i-1,j-2)+w_e} \\ \frac{d(i,j)+S(i-1,j-1)}{S(i-1,j-1)+w_d} \\ \frac{d(i,j)+S(i-1,j)}{S(i-1,j)+w_s} \end{cases}$$

Here, w_e, w_s, w_d represent weights associated for the corresponding transition. The DP table for this problem is a matrix of size $(m \times n)$ where each cell of the table contains a tuple (s,t) where s and t are entries corresponding to the similarity matrix and transition matrix respectively. As suggested by above equations, the problem belongs to *1-Sided, Contiguous* subclass. The j^{th} cell in the first row of the DP table is initialized to $(d(1,j),0)$.

Figure 7(a) shows the performance of different implementations (CPU parallel, GPU and Framework) of *NS-DTW problem* for different sizes of DP table.

B. Checkerboard Problem

Checkerboard problem is a standard problem which can be solved using `HLDP` characterization. In this problem, we have a grid of size $n \times n$ and $c(i,j)$ represents the associated $cell_{i,j}$ of the grid. The goal is to find the shortest path from any cell in the first row to any cell in the n^{th} row. Here the shortest path means the path where sum of the costs of the visited cells is minimum. The paths under consideration have to maintain the constraint that from a given cell, one can visit the neighboring cells that are diagonally left forward, diagonally right forward, or straight forward. For example, from $cell_{i,j}$, the path can go to one of $cell_{i-1,j-1}$, $cell_{i-1,j}$ and $cell_{i-1,j+1}$.

According to the description given above, the shortest distance to reach $cell_{i,j}$ can be computed by following function.

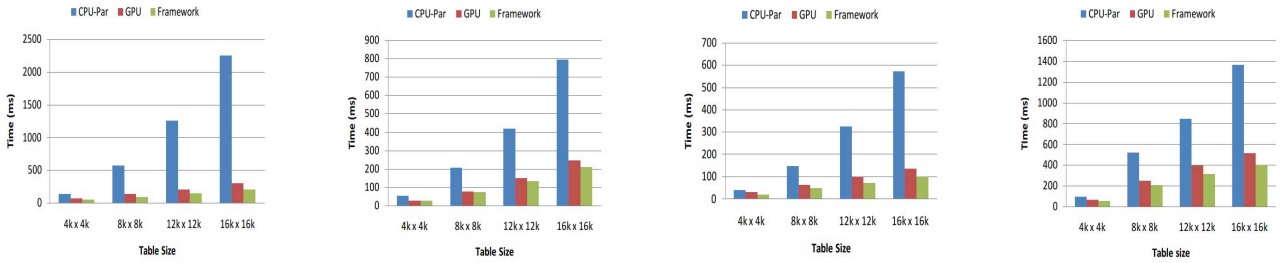
$$f(i,j) = \begin{cases} \infty, & \text{if } (j<1) \text{ or } (j>n) \\ c(i,j), & \text{if } (i=1) \\ min \begin{cases} f(i-1,j-1)+c(i,j), \\ f(i-1,j)+c(i,j), \\ f(i-1,j+1)+c(i,j) \end{cases} & \text{otherwise} \end{cases}$$

This function suggests that checkerboard problem belongs to 2-Sided Contiguous subclass. We initialize each cell in the DP table with its associated cost.

Figure 7(b) shows the performance of different implementations (CPU parallel, GPU and Framework) of *checkerboard problem* for different sizes of DP table. Apart from kernel setup time, we have additional overheads of pinned memory access and data transfer (2-way). These overheads are more than actual execution time for smaller table sizes. However, as the table size grows, work partitioning improves the performance of the heterogeneous algorithm over a pure GPU implementation.

C. 0/1 Knapsack Problem

One of the classical problems that can be approached using the `HLDP` characterization is the 0/1 Knapsack Problem. Given m items with weights $(w_1, w_2, w_3, \ldots, w_m)$ and values $(v_1, v_2, v_3, \ldots, v_m)$, the goal is to select items such that sum of their weights are less than or equal to W and gain (sum of



(a) (b) (c) (d)

Fig. 7. Running time (in milliseconds) for different table sizes (n x n) for (a) NS-DTW problem, (b) the checkerboard problem, (c) the 0/1 Knapsack problem, and (d) the multistage-networking problem, respectively.

their values) is maximum. This problem is solved by dynamic programming approach where we construct a table of size $(m = 1) \times (W + 1)$. $cell_{i,j}$ of this table represents maximum possible gain with weight less than or equal to j using first i items. This solution is given by following function.

$$f(i,j) = \begin{cases} f(i-1,j), & w_i > j \\ max \begin{cases} f(i-1,j) \\ v_i + f(i-1, j-w_i) \end{cases} & \text{otherwise} \end{cases}$$

According to this function, this problem belongs to 1-Sided Non-contiguous subclass. Each cell in the DP table is initialized with 0. Figure 7(c) shows the performance of different implementations (CPU parallel, GPU and Framework) of *0/1 knapsack problem* for different sizes of DP table.

D. A multi-stage networking problem

Consider a network where total $(m \times n)$ computers are connected in the following way. There are m stages having n computers each. All n computers in a given stage are independent and no connections exist within the same stage. Each computer in stage i is connected to at-most k (constant) computers in stage $(i + 1)$. The communication cost is negligible for these connections.

A task (consisting of m dependent subtasks) is to be scheduled on this network such that $subtask_i$ can be scheduled on any computer in $stage_i$. $subtask_i$ is dependent on $subtask_{i-1}$. $cost_{i,j}$ represents the time required to finish the $subtask_i$ on j^{th} computer of $stage_i$. The goal is to find the least possible time for task completion.

According to the description given above, the least possible time can be computed by following function.

$$f(i,j) = \begin{cases} \infty, \text{if } (j < 1), \text{ or } (j > n) \\ cost(i,j), \text{if } (i = 1) \\ min \begin{cases} f(i-1, conn_{i,j,1}) + cost(i,j), \\ f(i-1, conn_{i,j,2}) + cost(i,j), \\ \vdots \\ f(i-1, conn_{i,j,k}) + cost(i,j) \end{cases} & \text{otherwise} \end{cases}$$

Here $conn_{i,j,k}$ is the column index of the k^{th} computer connected to $computer_{i,j}$.

Since connection list of each computer has no correlation to each other, this problem belongs to *Non-Uniform HLDP* class. We initialize each cell in the DP table with its associated cost.

Figure 7(d) shows the performance of different implementations (CPU parallel, GPU and Framework) of above problem for different sizes of DP table.

VII. Conclusions

In this paper, we proposed a framework for Horizontally Local Dynamic Programming problems on heterogeneous systems. We demonstrated different strategies for efficiently solving different cases of `HLDP` problems. Our framework can ease the process of developing efficient programs for the targeted class of problems.

References

[1] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley, Technical Report No. UCB/EECS-2006-183, 2006.

[2] V. Boyer, D. El Baz, and M. Elkihel, "Solving knapsack problems on gpu," Computers and Operations Research, vol. 39, pp. 42-47, 2012.

[3] Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: SPAA, pp. 207-216. ACM (2008).

[4] Rezaul Alam Chowdhury, Hai-Son Le, Vijaya Ramachandran: Cache-Oblivious Dynamic Programming for Bioinformatics. IEEE/ACM Trans. Comput. Biology Bioinform. 7(3): 495-510 (2010)

[5] El Baz D, Elkihel M. Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem. Journal of Parallel and Distributed Computing 2005;65:74-84

[6] Gerash TE, Wang PY. A survey of parallel algorithms for one-dimensional integer knapsack problems. INFOR 1993;32(3):163-86

[7] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition, Addison-Wesley, 2003.

[8] " Carl Kingsford. Dynamic Programming: Subset Sum & Knapsack. https://www.cs.cmu.edu/ ckingsf/class/02713-s13/lectures/lec15-subsetsum.pdf (March. 2013)"

[9] Rajesh Kumar, Kishore Kothapalli, A Novel Heterogeneous Parallel Framework for Local Dependency Dynamic Programming Problems, IEEE- IPDPSW PLC 2015.

[10] Lou DC, Chang CC. A parallel two-list algorithm for the knapsack problem. Parallel Computing 1997;22:1985-96.

[11] Gautam Varma Mantena, Sivanand Achanta, Kishore Prahallad, Query-by-Example Spoken Term Detection using Frequency Domain Linear Prediction and Non-Segmental Dynamic Time Warping. IEEE/ACM Transactions on Audio, Speech & Language Processing 22(5): 944-953 (2014)