A Novel Heterogeneous Framework for Local Dependency Dynamic Programming Problems

Rajesh Kumar*, Kishore Kothapalli†
Center for Security, Theory and Algorithmic Research (CSTAR),
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India, 500 032
Email: *rajesh.kumar@research.iiit.ac.in, †kkishore@iiit.ac.in

***Abstract*—Dynamic programming is a well known technique to solve combinatorial search and optimization problems. LDDP (Local Dependency DP) problems are those DP problems where each update to an entry in the DP table is determined by the contents of its neighboring cells. The amount of parallelism available for a given LDDP problem may vary depending on the position and number of neighboring cells involved in the computation.**

In this paper, we develop a parallel heterogeneous (CPU+GPU) framework for solving LDDP problems. We use GPU and CPU specific optimizations and investigate if the given problem is a good candidate for heterogeneous computation. We also devise techniques to distribute the workload between CPU and GPU. We show how our framework can be used by considering several case study problems.

I. Introduction

Dynamic Programming(DP) is a powerful technique used for efficiently solving a vast set of complex optimization problems. It is widely used in areas of scheduling, string editing, packaging, bioinformatics, and inventory management [14]. Because of its impact and applicability across a wide range of domains, it is believed that DP will retain its importance in the future for science and engineering. This importance makes it one of the Berkeley 13 dwarfs [3].

Dynamic programming aims at solving complex problems by dividing them into simpler subproblems. For dynamic programming to be applicable, the problem must exhibit two key properties, *optimal substructure* and *overlapping subproblems*.

- A problem exhibits the *optimal substructure* property if its solution can be obtained efficiently by combining the optimal solutions to its subproblems.
- A problem exhibits the *overlapping subproblems* property if the space of its subproblems is small. That is, problem can be decomposed into subproblems which are reused several times.

Dynamic programming problems can be approached either in *top-down* way or *bottom-up* way. In the top-down approach, we can formulate solution to a DP problem by recursively using the solution to its subproblems. Since the subproblems are overlapping, we can easily store the solutions of already computed subproblems into a table and use them later. In the bottom-up approach, we solve the subproblems first, and use these solutions to build solutions to bigger subproblems. Usually an iterative approach is followed to construct solutions to bigger subproblems using solutions to smaller subproblems.

A special class of DP problems called LDDP (Local Dependency DP) problems are ones in which update to an entry in the DP table is determined by the contents of neighboring cells [8]. Many non DP problems also mimic local dependency nature of LDDP problems. We collectively classify these local dependency problems (DP and non DP) as LDDP-Plus Problems. Examples of this class of problems are common in many domains like speech processing (Dynamic Time Warping Algorithm [2]), bioinformatics (longest common subsequence, pairwise sequence alignment with affine gap cost [8]), image (Floyd-Steinberg dithering [12]) and the like.

Every LDDP-Plus problem has a table associated with it, which can be filled iteratively (bottom-up approach) by using the corresponding formula. The general form of the LDDP-Plus formula can be given as below where the function f depends on the problem.

$$table[i][j] = f\begin{pmatrix} table[i][j-1], table[i-1][j-1], \\ table[i-1][j], table[i-1][j+1] \end{pmatrix}$$

The number and position of neighboring cells involved in the function will determine the extent of parallelism available for the given problem and the complexity involved in parallelization.

In this work, we categorize LDDP-Plus problems into four different categories. We design and implement heterogeneous algorithms for each of these four categories of problems as a complete framework. Our framework can act as a software tool that can enhance the productivity of programming modern heterogeneous systems. Using our framework, users can readily create solutions for problems that follow local dependency dynamic programming approach by just providing the function that dictates the dependency pattern.

Since parallelism profiles (degree of parallelism v/s time plot) for the four categories are significantly different from each other, we design separate heterogeneous execution strategies for each one of them. Our focus here is to give the right amount of work to right computational unit at the right time. One crucial thing, apart from a good workload division scheme, in a heterogeneous setup is to hide (or at least minimize) the data transfer latency between the host (CPU) and the device (GPU). For this purpose, we propose a pipeline

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.25

based data transfer scheme for hiding data transfer latency between the CPU and the GPU. However, due to complex data patterns, sometimes this pipeline scheme is not applicable. We identify those cases and propose a pinned memory based inexpensive data transfer scheme for them.

We summarize our main technical contributions of this work as follows.

- We categorize LDDP-Plus problems into four categories on the basis of their parallelism profiles.
- We propose a heterogeneous framework that includes high level support for each category of problems. A user of the framework needs to supply only the problem dependent function f.
- We introduce optimizations in the framework such as efficient workload division schemes, pipelining to hide communication costs, and coalescing to improve memory access.
- We also show three case studies that indicate how to use our framework along with results on two different heterogeneous computing platforms.

It is important to note that every LDDP-Plus problem might have potential for some problem specific optimizations. However, we are just interested in the problem independent optimizations. Our aim is to achieve good performance for all (LDDP-Plus) problems against excellent performance for a specific problem.

Our consideration of heterogeneous algorithms is motivated by the fact that the current architectural trend in parallel computing is towards a heterogeneous collection of devices involving CPUs and accelerators such as GPUs and Intel Xeon-Phi. Hence, the design and development of heterogeneous algorithms aimed at commodity heterogeneous computing platforms are of immense research interest. Heterogeneous algorithms for a variety of problems from domains such as sorting [4], graph algorithms [5] [13], sparse matrix computations [18], are reported in recent literature.

A. Related Work

Local dependency Dynamic programming has direct applications in different domains. It is thus not surprising that a lot of research attention is devoted in parallelizing LDDP problems. Early works have largely focused on problem specific solutions. Fast problem specific algorithms like parallel bit vector algorithm (for longest common subsequence problem) has reached from its primitive parallel solutions (by Allison and Dix [1]) to very fast GPU based solutions proposed by Kloetzli et al. [17] and Kawanami et al. [16]. Deshpande et al. [11] presented heterogeneous implementation of error diffusion dithering (which is a non-DP local dependency problem).

Focusing on generic solution to LDDP class of problems, one of the first notable works is that of Chowdhury et al. [8]. They present a generic CMP(cache-efficient chip multiprocessor) algorithm with an associated tiling sequence. For each type of CMP (D-CMP, S-CMP, or Multicore), they specify the tiling parameter and parallel schedule to ensure good performance wrt parallelism and cache efficiency. Chowdhury et al. [9] also developed cache-oblivious algorithms for dynamic programming problems in bioinformatics.

Solution to LDDP problems in unreliable memory was proposed by Caminiti et al. [7]. Bille et al. [6] presented cache oblivious algorithms to suit string comparison based LDDP problems (like LCS) in which at most three neighboring cells are involved in computing the value of the current cell. All these solutions are CPU based.

Cuencaa et al. [10] presents heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems.

B. Organization of the Paper

In the rest of the paper, we first describe some background material in Section II. Section III gives the details of heterogeneous framework. We discuss about some optimization techniques in Section IV. Section V presents some implementation details. Case study of different types of problems is presented in Section VI followed by concluding remarks in Section VII.

II. Preliminaries

The class of *LDDP-Plus* problems is characterized by a function which considers values of previously computed adjacent cells to update each position in a k ($k \geq 2$) dimensional table. For simplicity, we deal with 2 dimensional LDDP-Plus problems in this paper.

Every cell, apart from boundary cells, in a 2 dimensional table is surrounded by 8 neighboring cells. So the value to be filled in $cell_{i,j}$ of the table is defined by following function.

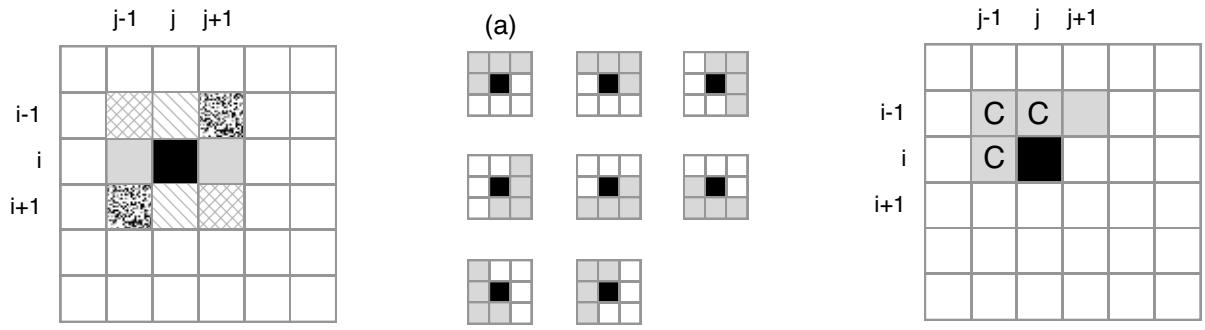
$$cell_{i,j} = f(cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1}, cell_{i,j+1}, cell_{i+1,j+1}, cell_{i+1,j}, cell_{i+1,j-1}, cell_{i,j-1})$$

We say that two cells are said to be conflicting each other with respect to $cell_{i,j}$ if the following conditions hold.

- both the cells are neighbors of $cell_{i,j}$ AND
- a straight line drawn through them passes through $cell_{i,j}$.(See Figure 1(a)).

Usually, the function f in LDDP-Plus problems does not vary with respect to the cell that is being filled. Thus, the process of filling the table cannot proceed if the value to be filled in $cell_{i,j}$ is dependent on the conflicting cells due to cyclical dependencies. This implies that the value to be filled in $cell_{i,j}$ cannot be dependent on more than four neighboring cells. Also none of these four cells should be pairwise conflicting.

To categorize LDDP-Plus problems, we select a set of four non-conflicting cells as a *representative set*. The elements of this set are called the *representative cells*. The choice of a representative set among all sets having four non-conflicting cells does not impact our categorization or our framework implementation. They can be viewed as similar sets (dependency structure) by appealing to symmetry. For our paper, for a cell with indices i and j, we use following set as our representative set.



(a) Neighboring Cells and Conflicting Cells (b) Representative Sets (c) Contributing Cells

Fig. 1. (a) The black cell represents the cell under consideration ($cell_{i,j}$). All the 8 cells surrounding $cell_{i,j}$ are neighboring cells (represented by shades). Pairs having same shading types are conflicting to each other. (b) Grey cells represent representative cells. There are 8 representative sets possible. We choose the set marked with 'a' as our representative set. (c) Representative cells (grey) marked with 'c' represent contributing cells for longest common subsequence problem.

$$\text{RS}(i,j) = \{\ cell_{i,j-1}, cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1}\ \}$$

Depending on the function f that is problem specific, one or more of representative cells decides the value to be filled in $cell_{i,j}$. These cells are called *contributing cells*. The set of contributing cells is called *contributing set*.(See Figure 1(c)).

It can be now noted that while our framework uses a common representative set across the four categories of LDDP-Plus problems, changes to the contributing set is what induces the four categories in LDDP-Plus problems as follows. Every LDDP-Plus problem is associated with a function which defines how the value of $cell_{i,j}$ is filled in a given iteration. This function is characterized by number and position of contributing cells. This function takes one or more cells (contributing cells) from the representative set as input and returns the value to be filled in $cell_{i,j}$ as output.

$$cell_{i,j} = f(cell_{i,j-1}, cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1})$$

A. A Brief Overview of our Experimental Platform

In this section, we briefly describe the heterogeneous platforms used in our experiments. We use two different heterogeneous platforms and both these platforms are a combination of Intel multicore CPUs and Nvidia GPUs. One (labeled *Hetero-High*) is composed of server class hardware, which is typically the one that is used for developmental purposes and the other (labeled *Hetero-Low*), represents commonly used commodity desktop and laptops configurations.

1) Hetero-High: It consists of an Intel i7 980 CPU and an Nvidia Tesla K20 GPU. The Intel i7 980 has six cores with each core running at 3.33 GHz. The i7 980 can handle twelve logical threads (with hyper threading). Tesla K20 is based on Nvidia's Kepler microarchitecture with 13 streaming multiprocessors (SMX) with each having 192 cores for a total of 2496 compute cores.

2) Hetero-Low: It is a relatively low end setup. It consists of an Intel i7-3632M CPU and an Nvidia GeForce GT650M GPU. The Intel i7-3632M has four cores with each core running at 2.2 GHz. The i7 980 can handle eight logical threads (with hyper threading). GT650M has 2 streaming multi-processors (SMX) with each having 192 cores for a total of 384 compute cores.

We use OpenMP specification 3.0 for thread creation on the CPU. To program the GPU we use the CUDA API Version 5.0. The CUDA API Version 5.0 supports asynchronous concurrent execution model so that a GPU kernel call does not block the CPU thread that issued this call. This also means that execution of CPU threads can overlap a GPU kernel execution.

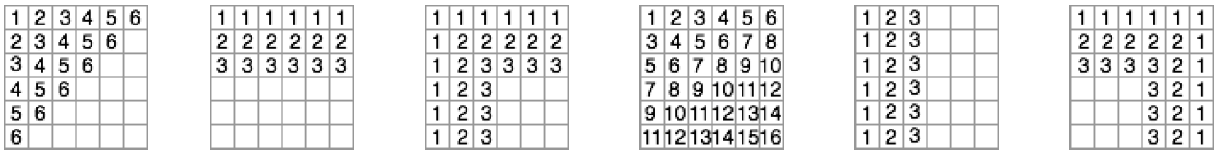
III. Framework Description

In this section, we explain the categorization we achieve for LDDP-Plus problems. Recall that each such LDDP-Plus problem can be associated with its contributing cells that dictate the inputs to the function f. As shown in Section II, there are at most four contributing cells for any LDDP-Plus problem. Therefore, depending on the position and number of contributing cells, a total of $(2^4 - 1) = 15$ different types functions can be formed as shown in Table I. All these 15 combinations of contributing cells can be mapped into six types of patterns as shown in Table I. See also Figure 2 for an illustration.

Of the six patterns patterns Vertical and Horizontal are symmetric in nature. Similarly, patterns Inverted-L and mirrored Inverted-L are also symmetric as shown in Figure 2. Therefore, these four patterns can be reduced to two patterns, Horizontal and Inverted-L. The other two patterns can be addressed by appealing to symmetry. Thus, we are left with only four distinct patterns for LDDP-Plus problems.

The input to our framework is a user-defined LDDP Plus function, which takes one or more of representative cells as input, and returns the value to be filled in $cell_{i,j}$ as output. On the basis of number and position of contributing cells, the framework maps the function to a pattern. The selected pattern describes which cells will be processed in parallel in a given iteration.

Based on the parallelism profile of the selected pattern, the



(a) Anti-Diagonal (b) Horizontal (c) Inverted-L (d) Knight-Move (e) Vertical (f) mInverted-L

Fig. 2. Pattern Types. For a given pattern, all cells marked with number i can be processed in the i^{th} iteration. Cells marked with same number can be processed in parallel.

$cell_{i,j-1}$	$cell_{i-1,j-1}$	$cell_{i-1,j}$	$cell_{i-1,j+1}$	**Pattern**
N	N	N	Y	mInverted-L
N	N	Y	N	Horizontal
N	N	Y	Y	Horizontal
N	Y	N	N	Inverted-L
N	Y	N	Y	Horizontal
N	Y	Y	N	Horizontal
N	Y	Y	Y	Horizontal
Y	N	N	N	Vertical
Y	N	N	Y	Knight-Move
Y	N	Y	N	Anti-diagonal
Y	N	Y	Y	Knight-Move
Y	Y	N	N	Vertical
Y	Y	N	Y	Knight-Move
Y	Y	Y	N	Anti-diagonal
Y	Y	Y	Y	Knight-Move

TABLE I
DIFFERENT CONTRIBUTING SETS AND CORRESPONDING PATTERN

framework divides the work between the CPU and the GPU. In a given iteration, both the CPU and the GPU may participate in computing the values for disjoint set of cells. The value computed by the CPU (or GPU) might be needed by the GPU (or CPU) in subsequent iterations. The framework takes care of efficient data transfer between the CPU and the GPU.

In the following subsections, we describe our algorithm execution strategies in a heterogeneous setup for the four different patterns identified earlier.

A. Anti-diagonal Pattern

If the contributing set is either $\{cell_{i,j-1}, cell_{i-1,j-1}, cell_{i-1,j}\}$ or $\{cell_{i,j-1}, cell_{i-1,j}\}$, we can compute the values of the cells across an anti-diagonal in parallel. The degree of parallelism increases with each iteration until the main diagonal of the table is reached. After that the degree of parallelism starts decreasing. In a pure GPU (or CPU) approach, we can assign one thread (or block of threads) to each cell (or a set of cells) of a given anti-diagonal. For diagonals with a low degree of parallelism, the overhead of parallelism can outweigh the benefits of a heterogeneous approach. We therefore let the CPU do entire work in such regions. However, as the degree of parallelism increases, we can distribute the work between the

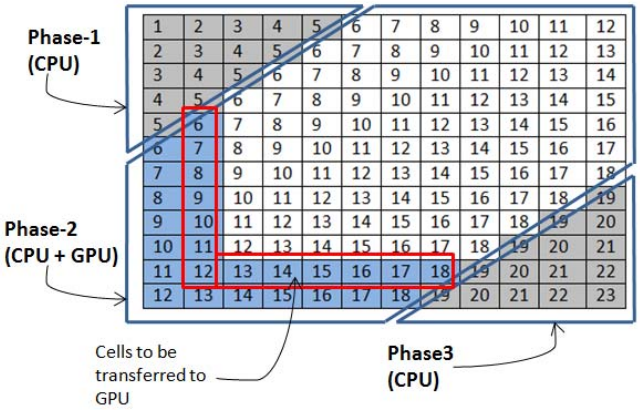


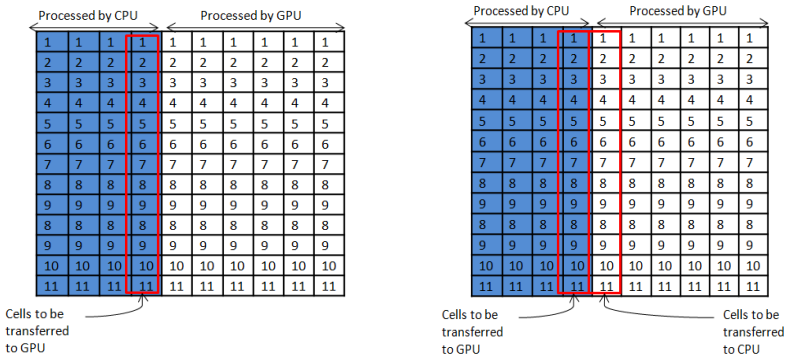
Fig. 3. Heterogeneous execution of Anti-diagonal pattern problems. Cells processed by CPU is colored in grey (low work region) and blue (in high work region). Uncolored cells are processed by GPU. In Phase-2, we need to transfer boundary cells (red boundary) to GPU. To process i_{th} anti-diagonal, GPU needs boundary cells from last two anti-diagonals.

CPU and the GPU. As shown in Figure 3, the heterogeneous algorithm can be completed in following three phases. In the following, t_{switch} and t_{share} are parameters whose values will be specified later.

Phase-1:
For the first t_{switch} iterations, we let the CPU process all cells lying across the corresponding anti-diagonal. t_{switch} is the threshold (number of iterations) crossing which the program switches from a low work region to a high work region.

Phase-2:
For the next $(totalNumberOfIterations - 2 * t_{switch})$ iterations, we distribute the work between the CPU and the GPU. We assign first t_{share} cells of the corresponding anti-diagonal to the CPU and rest to the GPU. Note that because of the dependency created by $cell_{i,j-1}$ and $cell_{i-1,j-1}$, the GPU needs the values of these cells (computed by the CPU) lying on CPU-GPU boundary to proceed. So in each iteration of this phase, data transfer between the CPU and the GPU is needed.

Phase-3:
For next t_{switch} iterations, we let the CPU operate on all the cells of an anti-diagonal (like phase-1).



(a) Case-1 (b) Case-2

Fig. 4. Heterogeneous execution of horizontal pattern problems. Cells processed by the CPU is colored in blue. Uncolored cells are processed by GPU. In Case-1, We need to transfer boundary cells(red boundary) to GPU. To process i_{th} row, the GPU needs boundary cells from last row. In Case-2, We need to transfer boundary cells (red boundary) both ways (CPU to GPU and GPU to CPU).

Such a strategy of using the CPU alone or the CPU and the GPU over varying degree of parallelism is a popular strategy that is used also in [15], [11] among others.

B. Horizontal Pattern

In this type of pattern, the value of $cell_{i,j}$ is dependent only on the values of cells from the $(i-1)$th row. As can be noted from Table I, the possible contributing sets for this pattern are $\{cell_{i-1,j-1}, cell_{i-1,j}\}$, $\{cell_{i-1,j}, cell_{i-1,j+1}\}$, $\{cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1}\}$, $\{cell_{i-1,j-1}, cell_{i-1,j+1}\}$ and $\{cell_{i-1,j}\}$.

One important characteristic of this pattern is that the degree of parallelism is constant across all the iterations. In a pure GPU (or CPU) approach, we can assign one thread (or block of threads) to each cell (or a set of cells) of a row. Since enough amount of work is available in all iterations, we can distribute the workload between the CPU and the GPU from the first iteration. Also, since the amount of work available is same in all the iterations, a similar work division strategy can be followed in all the iterations. As shown in Figure 4, the heterogeneous algorithm can be completed in one phase. In the following, t_{share} is a parameter whose value will be specified later.

Phase-1:

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. Depending on the contributing set, one-way (CPU to GPU or GPU to CPU) or two-way (CPU to GPU and GPU to CPU) data transfer is required in each iteration. However, for the contributing set $\{cell_{i-1,j}\}$, data transfer is not needed.

Data Movement:

- Case-1: If the contributing set is $\{cell_{i-1,j-1}, cell_{i-1,j}\}$ or $\{cell_{i-1,j}, cell_{i-1,j+1}\}$, one-way data transfer is enough.
- Case-2: If the contributing set is $\{cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1}\}$ or $\{cell_{i-1,j-1}, cell_{i-1,j+1}\}$, two-way data transfer is needed.

C. Inverted-L Pattern

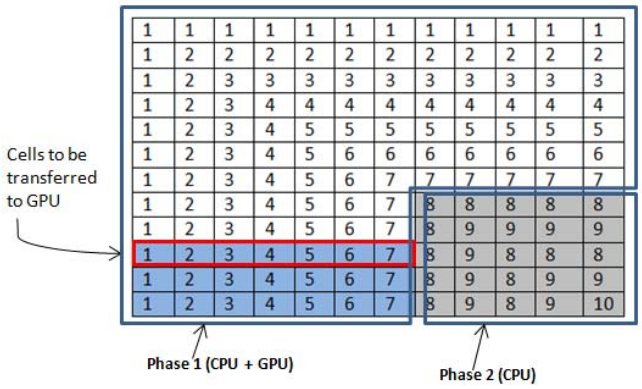


Fig. 5. Heterogeneous execution of Inverted-L pattern problems. Cells processed by the CPU is colored in grey (low work region) and blue (in high work region). Uncolored cells are processed by GPU. In Phase-1, we need to transfer boundary cells (red boundary) to GPU. In the i_{th} iteration, GPU needs boundary cells (having red boundary) from last one iteration (inverted-L shape).

If the contributing set is $\{cell_{i-1,j-1}\}$, this pattern is followed. In each iteration, we process cells lying across an inverted-L shape. The size of inverted-L decreases with each iteration. Hence, the degree of parallelism decreases with time.

In a pure GPU (or CPU) approach, we can assign one thread (or block of threads) to each cell (or a set of cells) lying on corresponding inverted-L. Since enough amount of work is available in early iterations, we can distribute the workload between the CPU and the GPU from first iteration. Once the amount of work reduces to a threshold, we can allow CPU to take full control. As shown in Figure 5, the heterogeneous algorithm can be completed in two phases. In the following, t_{switch} and t_{share} are parameters whose values will be specified later.

Phase-1:

For the first $(totalNumberOfIterations - t_{switch})$ iterations, we distribute the work between the CPU and the GPU. We assign the first t_{share} cells of the corresponding inverted-L to the CPU and rest to the GPU. Note that because of the dependency created by $cell_{i-1,j-1}$, GPU needs the values of the cells (computed by CPU) lying on the CPU-GPU boundary to proceed. So in each iteration of this phase, data transfer between the CPU and the GPU is needed.

Phase-2:

When only t_{switch} iterations are left, we switch from a high work region to a low work region. Hence we allow the CPU to operate on all the cells of the corresponding inverted-L.

Alternatively, this class of problems can be solved using horizontal pattern (Case-1).

D. Knight-Move Pattern

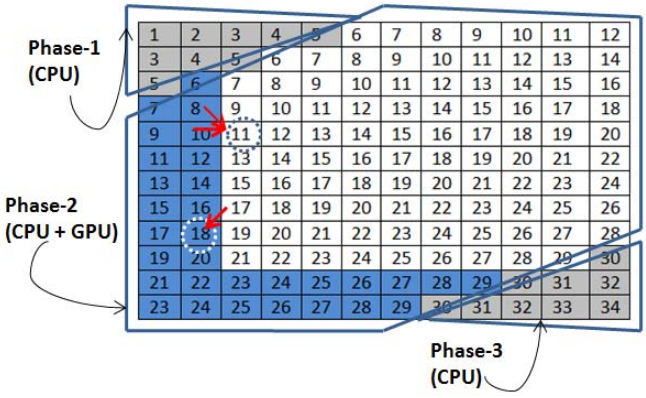


Fig. 6. Heterogeneous execution of the Knight-Move pattern problems. Cells processed by the CPU are colored in grey (low work region) and blue (in high work region). Uncolored cells are processed by GPU. In Phase-2, two-way transfer of boundary cells is needed. In the figure, to compute cell 11 (GPU boundary cell, circled in blue dots), GPU need values 8 and 10 (red arrows) from the CPU. To compute cell 18 (CPU boundary cell, circled in blue dots), the CPU need value 17 (red arrow) from the GPU.

If the contributing set contains both $cell_{i-1,j+1}$ and $cell_{i,j-1}$, then the knight-move pattern is followed. The parallelism profile of problems belonging to this pattern resembles that of anti-diagonal pattern based problems. i.e. the degree of parallelism increases with each iteration until half the iterations are over. After that the degree of parallelism starts decreasing.

So, like the anti-diagonal pattern (see Figure 3), the heterogeneous algorithm for the knight-move pattern problems can be completed in three phases as shown in Figure 6. In the following, as earlier, t_{switch} and t_{share} are parameters whose values will be specified later.

Phase-1:

For the first t_{switch} iterations, we let the CPU process all the cells. t_{switch} is the threshold (number of iterations) crossing which the program switches from a low work region to a high work region.

Phase-2:

For the next $(totalNumberOfIterations - 2 * t_{switch})$ iterations, we distribute the work between the CPU and the GPU. In each iteration, we assign first t_{share} cells to the CPU and rest to the GPU. Note that anti-diagonal pattern requires one-way data transfer only. But knight-move pattern requires two-way data transfer.

Phase-3:

For next t_{switch} iterations, we let the CPU operate on all the cells.

This heterogeneous execution scheme is similar to the one proposed by Deshpande et al. [11] for implementation of Floyd Steinberg Dithering.

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

On the CPU, creating large number of threads is not a good choice for standard reasons such as the overhead of thread creation and context switching. So, for processing on the CPU, we create a few heavy-weight threads where each thread is responsible for processing a group of cells (one or more blocks/sub-blocks).

On the other hand, to exploit massively parallel architecture of the GPU, creating a large number of light-weight threads is the best choice. So, we follow thread per cell strategy on GPU.

B. Memory Coalescing on the GPU

Whenever a thread accesses the global memory, it always acts on a large chunk of memory at once even if the thread needs to access a small chunk. If other threads are accessing the contiguous memory locations, GPU can reuse the same large chunks for other threads. We make the access pattern coalesced by simply storing all the cells marked with the same number in Figure 2 together in a one dimensional array and maintaining non decreasing order. Thus, the way we store the table in our framework depends on the nature of accesses to the table over the four different categories of LDDP-Plus problems.

C. Managing Data Transfers

If both CPU and GPU are involved in processing, the need of data transfer arises. Table II shows the data transfer needs corresponding to each pattern. Based on whether one way data transfer is enough or not, we propose two different strategies for efficient data transfer.

1) Case-1: One Way Transfers: If we need to transfer the values only from the CPU to the GPU (or vice versa), we can use a *pipelining approach*. To explain this approach, we use the horizontal pattern as an example with the contributing set as $\{cell_{i-1,j-1}, cell_{i-1,j}\}$. To compute the value of $cell_{i,j}$ on the GPU boundary, we need boundary values (from the CPU), from previous iteration. In the first iteration GPU sits idle. In the mean time the CPU can compute the first row, and transfer the 0th row to GPU simultaneously. This simultaneous operation is possible using *CUDA Streams*. In the next iteration, while the CPU computes the 2nd row and transfers the 1st row to GPU, GPU can compute the 1st row. this process goes on till $(n+1)$th iteration where GPU computes value of nth row and the CPU sits idle.

2) Case-2: Two way transfers: The pipelining approach does not work if two way data transfer is needed. It is important to note that we only transfer a few cells. So, we use pinned memory which provides fast memory access if data size is small.

Pattern	1-way / 2-way
Anti-diagonal	1 way
Horizontal (case-1)	1 way
Horizontal (case-2)	2 way
Inverted-L	1 way
Knight-move	2 way

TABLE II
PATTERNS AND CORRESPONDING DATA TRANSFER NEED. 1 WAY- (CPU TO GPU OR GPU TO CPU), 2 WAY (BOTH WAYS).

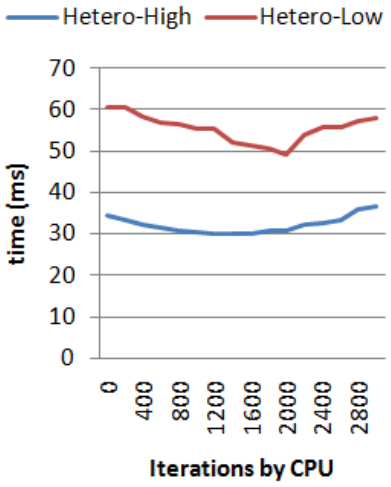


Fig. 7. Heterogeneous algorithm time for different number of iterations handled by the CPU in low-work region for longest common subsequence problem (DP table of size (4k x 4k)).

V. IMPLEMENTATION DETAILS

In the following subsections, we describe some implementation details along with some experiments to finalize our strategies.

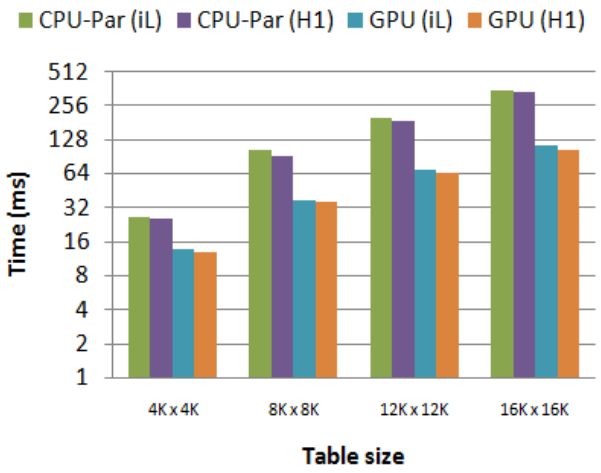


Fig. 8. Time comparison of inverted-L(iL) and horizontal case-1 (H1) pattern program on CPU and GPU.

A. Finding the Values of Parameters t_{switch} and t_{share}

Depending on the parallelism profiles, the heterogeneous framework divides the matrix into two types of segments (low work segment or high work segment). The number of iterations to be executed by the CPU in low work region is defined by t_{switch}. In the high work region, the number of cells processed by the CPU per iteration is defined by t_{share}. We obtain these values empirically as follows.

To know the optimal value of t_{switch}, we fix t_{share} to 0 and we run the algorithm for different values of t_{switch}. We plot the running time against different values of t_{switch}. As shown in Figure 7, this process generates a concave curve. The point corresponding to the minimum time on the curve indicates the optimal value of t_{switch}.

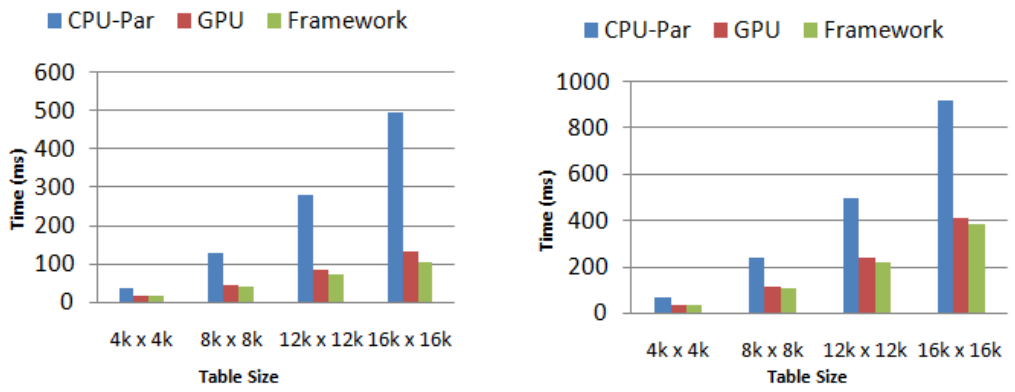
Now, we fix the value of t_{switch} to its optimal value, and we run the algorithm for different values of t_{share}. The point corresponding to the minimum time on the curve indicates the optimal value of t_{share}.

B. Horizontal Pattern (case-1) v/s inverted-L Pattern

To solve a problem whose contributing set is $\{cell_{i-1,j-1}\}$ or $\{cell_{i-1,j+1}\}$, we can either use the inverted-L pattern or case-1 of the horizontal pattern. Although, number of iterations in both the cases are same, uniformity in terms of number of cells per iteration and coalescing-friendly layout makes the horizontal pattern a better choice. Our experiments as described in Figure 8 support this choice. We have used the function $f(i,j) = max(cell_{i,j}, f(i-1,j-1)) + c$ for these experiments. Figure 9 shows the performance of different implementations (CPU, GPU and Framework) of a problem (characterized by the function $f(i,j) = min(f(i-1,j-1), f(i-1,j)) + c$) using case-1 of the horizontal pattern.

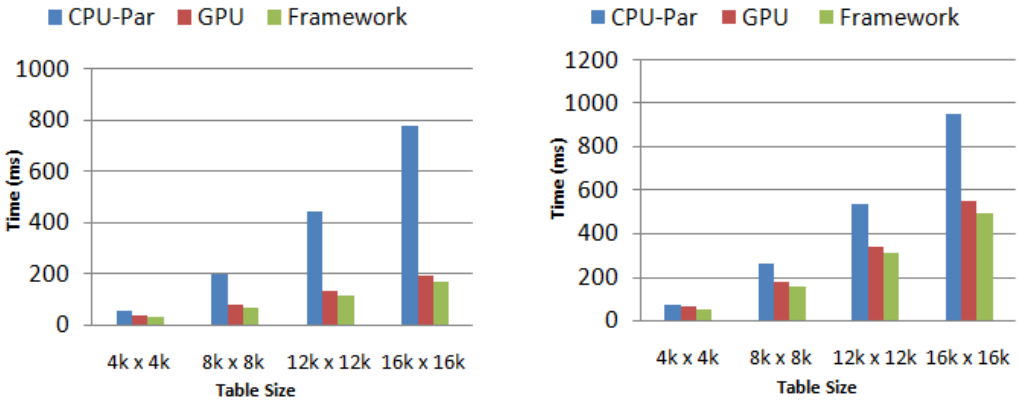
C. Using the Framework

For using this framework, a user has to provide the following.



(a) Hetero-High (b) Hetero-Low

Fig. 9. Time for different table sizes for Horizontal pattern case-1 on Hetero-High and Hetero-Low platforms.



(a) Hetero-High (b) Hetero-Low

Fig. 10. Time for different table sizes for Levenshtein Distance problem on Hetero-High and Hetero-Low platforms

1) Function f: A function defining how the value of $cell_{i,j}$ will be computed using neighboring cells and additional resources. The nature of this function identifies the category to which the problem belongs to.
2) Initialization: Since the initial value of the cells in the table for LDDP-Plus problems are problem dependent, the user has to provide the right initialization.

VI. Case Studies

We present the performance of our framework on three different problems for three different patterns. In the previous section, we have shown that the inverted-L pattern problems can be solved more efficiently by using the horizontal pattern (case-1). Also, we have already seen the performance of different implementations of the horizontal (case-1) pattern. So, in this section, we are ignoring these two patterns (inverted-L and horizontal case-1).

A. Levenshtein Distance using an Anti-diagonal Pattern

The Levenshtein distance [19] between two sequences is a metric which gives the minimum number of single character edits needed to change one sequence into the other. For any two sequences a and b, this metric can be calculated by following function.

$$f(i,j) = \begin{cases} max(i,j), & \text{if } min(i,j) = 0 \\ f(i-1,j-1), & \text{if } (a_i = b_j) \\ min \begin{cases} f(i-1,j)+1, \\ f(i,j-1)+1, \\ f(i-1,j-1)+1 \end{cases} & \text{otherwise} \end{cases}$$

This function suggests that the value of $cell_{i,j}$ is dependent on $cell_{i,j-1}$, $cell_{i-1,j-1}$ and $cell_{i-1,j}$ (in the 2-dimensional DP table). Hence, it follows the anti-diagonal pattern.

For two strings of length m and n, the size of the DP table is (m + 1) x (n + 1). All the cells in this table are initialized to 0.

Figure 10 shows the performance of different implementations (CPU parallel, GPU, Framework) of *Levenshtein Distance* for different problem sizes. Due to existence of low work region in the start and towards the end, the use of CPU improves the performance of heterogeneous algorithm over a pure GPU implementation (even for small table sizes). As the table size increases, the difference between execution times of

GPU and heterogeneous implementation becomes remarkable. This is attributable to the fact that, with increase in input size, the execution time becomes much higher than the kernel setup time.

B. Floyd Steinberg Dithering using the Knight Move pattern

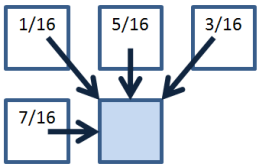


Fig. 11. Data dependency in Floyd Steinberg Dithering. $cell_{i,j}$ cannot be processed unless it receives the errors (scaled by factors shown in the figure) from its neighboring cells.

In this algorithm, for each pixel (i.e. $cell_{i,j}$) of the output image, the nearest color is calculated by using the threshold. Then the error is calculated for that pixel by comparing it with the corresponding pixel in the input image. This error is scaled by a factor of 7/16, 3/16, 5/16 and 1/16 and the resulting values are forwarded to $cell_{i,j+1}, cell_{i+1,j-1}, cell_{i+1,j}, cell_{i+1,j+1}$ respectively. Due to this, we cannot proceed with the calculation of $cell_{i,j}$ unless values of $cell_{i,j-1}, cell_{i-1,j-1}, cell_{i-1,j}, cell_{i-1,j+1}$ are available (See Figure 11). Thus, following scheduling constraint is followed.

$$Time(i,j) > max \begin{cases} Time(i,j-1), Time(i-1,j-1), \\ Time(i-1,j), Time(i-1,j+1) \end{cases}$$

Each pixel (cell) in the image (table) is initialized to 0.

Figure 12 shows the performance of different implementations (CPU parallel, GPU and Framework) of *Floyd Steinberg Dithering* for different sizes of DP table. For smaller images, multicore CPU implementation performs better than the GPU implementation (because CPU is more suitable for small number of heavy-weight threads). So for smaller images, heterogeneous implementation works as good as pure multicore-CPU implementation. But for larger images, the GPU implementation performs better than the CPU implementation. Also, work sharing makes heterogeneous implementation better than the pure CPU/GPU implementation as the image size grows.

Since we are using the same approach (for knight move pattern) as proposed by Deshpande et al. [11] for Floyd Steinberg Dithering, our results are also similar to them. The small improvement in performance can be attributed to the use of a relatively high end GPU by us.

C. Checkerboard Problem using Case-2 of the Horizontal pattern

One of the standard problems that can be solved using the LDDP-Plus characterization is the checkerboard problem. In the checkerboard problem, we have a grid of size $n \times n$, and each cell in the grid has a cost, $c(i,j)$ associated with it. The problem is to find the shortest path, in terms of the cells visited, from any cell in the first row to any cell in the n^{th} row. The paths considered have to maintain the constraint that from a given cell, one can go to the neighboring cells that are diagonally left forward, diagonally right forward, or straight forward. For example, from cell indexed (i,j), the path can go to one of $(i-1,j-1)$, $(i-1,j)$ and $(i-1,j+1)$.

Given the above description, the shortest distance to reach $cell_{i,j}$ can be calculated 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 value of $cell_{i,j}$ is dependent on $cell_{i-1,j-1}$, $cell_{i-1,j}$ and $cell_{i-1,j+1}$ (in the 2-dimensional DP table). Hence, it follows the horizontal pattern (case-2).

Each cell in the DP table is initialized with the cost associated with it.

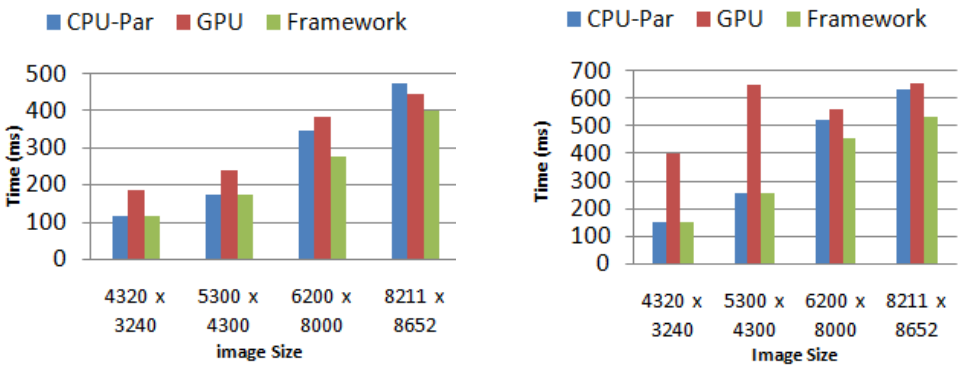
Figure 13 shows the performance of different implementations (CPU parallel, GPU and Framework) of *checkerboard problem* for different sizes of DP table. A low work region does not exist in this pattern. In high work region, apart from kernel setup time, we have additional overheads of pinned memory 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.

VII. Conclusions

In this paper, we proposed a framework for Local Dependency Dynamic Programming problems on a heterogeneous systems. For different cases of local dependency problems, we demonstrated whether the selection of heterogeneous approach is a good implementation strategy. As the input size grows, the idea of work sharing starts showing more positive impact on the implementation. It would be interesting to see how does a heterogeneous approach impact the implementation if the system has some other accelerators like Intel Xeon-Phi. Our framework can ease the process of developing efficient heterogeneous programs for the targeted class of problems.

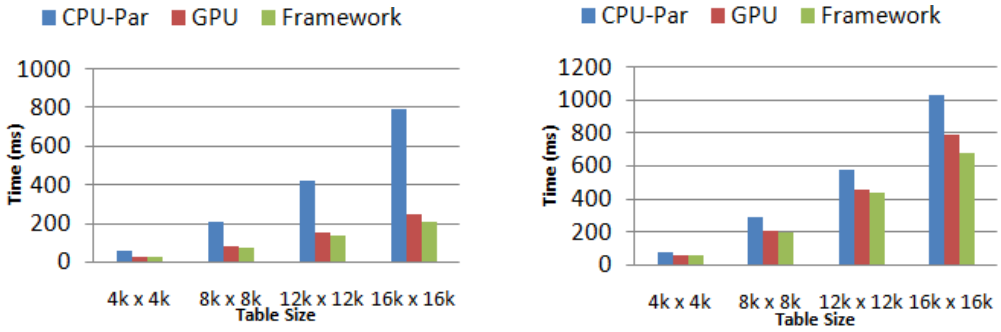
References

[1] L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. Inf. Process. Lett.,23(6):305310, Dec. 1986.
[2] Bin Amin T. and Mahmood I., Speech Recognition using Dynamic Time Warping, 2nd International Conference on Advances in Space Technologies, pp.74-79, 2008.
[3] 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.
[4] BANERJEE, D. S., SAKURIKAR, P., AND KOTHAPALLI, K. Fast, scalable parallel comparison sort on hybrid multicore architectures. In Proceedings of the third International workshop on Accelerators and Hybrid Exascale Systems (AsHES) (2013).
[5] BANERJEE, D. S., SHARMA, S., AND KOTHAPALLI, K. Work efficient parallel algorithms for large graph exploration. In Proc. Intl. Conf. on High Perf. Comp. (HiPC) (2013).



(a) Hetero-High (b) Hetero-Low

Fig. 12. Time for different image sizes for Floyd Steinberg Dithering on Hetero-High and Hetero-Low platforms



(a) Hetero-High (b) Hetero-Low

Fig. 13. Time for different table sizes for checkerboard problem on Hetero-High and Hetero-Low platforms

[6] Philip Bille, Morten Stckel. Fast and Cache-Oblivious Dynamic Programming with Local Dependencies. Language and Automata Theory and Applications. Lecture Notes in Computer Science Volume 7183, pp 131-142, 2012

[7] S. Caminiti, I. Finocchi, and E. G. Fusco. Local dependency dynamic programming in the presence of memory faults. In STACS, volume 9 of LIPIcs, pages 4556, 2011.

[8] Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: SPAA, pp. 207216. ACM (2008).

[9] 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)

[10] Javier Cuencaa, Domingo Gimnezb, Juan-Pedro Martnez. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. Parallel Computing. Volume 31, Issue 7, Pages 711735, July 2005

[11] Aditya Deshpande, Ishan Misra and P J Narayanan. Hybrid Implementation of Error Diffusion Dithering. In Proceedings of IEEE International Conference on High Performance Computing, HiPC, Dec 2011.

[12] R. Floyd and L. Steinberg An adaptive algorithm for spatial grey scale. Digest of the Society of Information Display, 1976.

[13] A. Gharaibeh, B. Costa, E. Santos-Neto, and M. Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In in Proc. of IEEE IPDPS (2013).

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

[15] Sungpack Hong, Tayo Oguntebi, Kunle Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. PACT '11: 20th International Conference on Parallel Architectures and Compilation Techniques, October 2011.

[16] K. Kawanami and N. Fujimoto. Gpu accelerated computation of the longest common subsequence. In Facing the Multicore - Challenge II, volume 7174 of LNCS, pages 8495. Springer Berlin, 2012.

[17] J. Kloetzli, B. Strege, J. Decker, and M. Olano. Parallel longest common subsequence using graphics hardware. In Proceedings of Eurographics conference on Parallel Graphics and Visualization, 2008.

[18] K. Matam, S. Indarapu, and K. Kothapalli. Sparse Matrix Matrix Multiplication on Hybrid CPU+GPU Platforms, in Proc. of HiPC, 2012

[19] Wagner, Robert A.; Fischer, Michael J. "The String-to-String Correction Problem", Journal of the ACM 21 (1): 168173, (1974)