
# X-TED: Massive Parallelization of Tree Edit Distance

Dayi Fan
The Ohio State University
fan.1090@osu.edu

Rubao Lee
Freelance
lee.rubao@ieee.org

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

## ABSTRACT

The tree edit distance (TED) has been found in a wide spectrum of applications in artificial intelligence, bioinformatics, and other areas, which serves as a metric to quantify the dissimilarity between two trees. As applications continue to scale in data size, with a growing demand for fast response time, TED has become even more increasingly data- and computing-intensive. Over the years, researchers have made dedicated efforts to improve sequential TED algorithms by reducing their high complexity. However, achieving efficient parallel TED computation in both algorithm and implementation is challenging due to its dynamic programming nature involving non-trivial issues of data dependency, runtime execution pattern changes, and optimal utilization of limited parallel resources.

Having comprehensively investigated the bottlenecks in the existing parallel TED algorithms, we develop a massive parallel computation framework for TED and its implementation on GPU, which is called X-TED. For a given TED computation, X-TED applies a fast preprocessing algorithm to identify dependency relationships among millions of dynamic programming tables. Subsequently, it adopts a dynamic parallel strategy to handle various processing stages, aiming to best utilize GPU cores and the limited device memory in an adaptive and automatic way. Our intensive experimental results demonstrate that X-TED surpasses all existing solutions, achieving up to 42x speedup over the state-of-the-art sequential AP-TED, and outperforming the existing multicore parallel MC-TED by an average speedup of 31x.

**PVLDB Reference Format:**
Dayi Fan, Rubao Lee, and Xiaodong Zhang. X-TED: Massive Parallelization of Tree Edit Distance. PVLDB, 17(7): 1683-1696, 2024.
doi:10.14778/3654621.3654634

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/Davis-Fan/X-TED.

## 1 INTRODUCTION

As a pivotal tool, the tree edit distance (TED) has found extensive applications in numerous domains dominated by tree-structured data. Specifically, in artificial intelligence (AI), TED is increasingly being adopted in AI-generated content verification and deep learning code generator [27, 29, 37, 43, 45]. With the popularity of large language models (LLMs), TED also plays an indispensable role in the training and performance measurement of these models [22, 49, 51, 72]. Moreover, TED boasts wide-ranging applications in bioinformatics [2, 10, 24, 56] and software engineering [23, 28, 34, 52, 70] as well.

Essentially, TED acts as a crucial metric to quantify the dissimilarity between two trees by representing the minimum cost of a sequence of editing operations, including inserting, deleting, and renaming nodes, which are required for transforming one tree into another. Therefore, it facilitates an effective comparison and analysis of hierarchical structures, such as parse trees, phylogenetic trees, and XML/HTML document trees, from various domains.

Unfortunately, the inherent high complexity of TED algorithms is well-known to cause increasingly long execution time as data volumes scale, making it unacceptable for many real-world applications. A basic and the most widely-used TED algorithm with the time complexity of $O(n^4)$ was published in 1989 [74], which originally introduced dynamic programming (DP) in TED and served as a basis for later optimized algorithms. Over the years, researchers have made dedicated efforts to improve TED algorithms by reducing the complexity. While recent optimized algorithms [9, 15, 46, 47] have achieved a cubic level complexity $O(n^3)$ for the worst case, the basic algorithm remains computationally competitive in certain common cases [7, 18]. Moreover, it has been proven that $O(n^3)$ is the theoretical lower bound for this class of algorithms [8, 15].

As Moore's Law approaches its limits and hardware accelerators like GPUs quickly emerge, massively parallel processing for TED is not only necessary but imperative. This inevitable shift enables us to effectively address the challenges posed by the ever-increasing volumes of data and the growing need for fast response time in TED computations. Therefore, it is critical to explore a parallel framework that is both efficient and feasible for TED algorithms.

However, parallelizing TED computation is far from being trivial either in algorithm design or in implementation. There have been limited research efforts on the development of parallel TED algorithms due to the intrinsic difficulties of this problem. A parallel version of the basic TED algorithm is also presented in the same paper [74], which is a comprehensive blueprint but poses three primary limitations when implemented on real-world parallel machines. (1) This parallel solution necessitates a huge memory space of $O(n^3)$ for data storage, which is a significant burden, even for trees with a moderate size of $n$ nodes. (2) Frequent synchronization operations among numerous processors are required, resulting in unacceptable execution overheads. (3) A highly imbalanced distribution of workloads causes a significant load-imbalance issue, seriously degrading execution performance. Despite several decades of research on TED, this parallel solution has not been practically implemented for real-world TED computations and applications.

The three systematic issues of the existing solution drive us to propose an entirely new and efficient parallel framework that truly enables feasible parallel accelerations for the TED computation.


Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654634

By closely analyzing the DP recurrence formula within the basic TED algorithm, we have uncovered that while the DP pattern is continually changing during computation, the data dependencies are essentially determined by the structures of two input trees. In TED computations, a substantial number of DP tables (2D matrices) are utilized to calculate and store intermediate results. The intricate data dependencies persistently exist among these DP tables. Considering each table as a single task and based on insights we obtained, we hereby develop an efficient algorithm that effectively determines the complex dependency relationships among tasks, **without** the need for intensive data analysis. Tables that are independent with each other can thus be computed concurrently and the required memory space has been dramatically reduced to $O(n^2)$. Also, the massive synchronization operations across tables have been eliminated. Another major challenge lies in the severe imbalanced workloads among tables. For each table, the number of entries required to compute may range from ones to millions. Therefore, we have designed a dynamic parallel strategy that allocates limited parallel resources to these unbalanced tasks in an adaptive and automatic way, which is crucial to achieve high performance.

In this paper, we begin by examining the intricate execution patterns of TED to gain insights and then identify several critical issues that hinder effective parallel processing of TED. We develop a massive parallel framework for TED computations and its implementation on GPU, which is called X-TED. For a given TED computation, X-TED applies a fast preprocessing algorithm to identify dependency relationships among millions of DP tables. Subsequently, TED parallel operations go through different processing stages. We develop a dynamic parallel strategy that handles table computations, aiming to best utilize many GPU cores and the limited device memory capacity in an adaptive and automatic way. Our contributions are summarized as follows.

- We delve into the intricate execution patterns of TED, identifying critical issues within the existing TED parallel algorithms. These findings motivate us to develop new and effective solutions.
- We develop and implement a fast and effective preprocessing algorithm to detect dependency relationships among all tables before parallel TED processing, without the need to construct and analyze the dependency graph.
- In any TED computation, we separate three distinctive stages during its parallel processing. We design an effective dynamic parallel strategy that best utilizes the available computing and memory resources across different stages.
- Putting all the above research efforts together, we construct a framework named X-TED for the parallel computation of TED and its implementation on GPU. Through extensive experiments and comprehensive comparisons, we show that X-TED surpasses all existing TED solutions. Specially, we demonstrate that X-TED achieves a remarkable speedup of up to 42x in comparison to the state-of-art sequential algorithm, and outperforms the existing multi-core parallel implementation by an average of 31x.

The paper is organized as follows. We introduce the basic TED algorithm in §2 and identify crucial issues in §3. We give an overview of our solution X-TED in §4. The detailed description of the efficient preprocessing algorithm and its mathematics proof is in §5. The implementation details of the dynamic parallel strategy are presented in §6. The overall experiment performance evaluation is given in §7. Finally, we review related work of TED in §8, and conclude our work with a summary in §9.

# 2 BACKGROUND

## 2.1 Tree Edit Distance

Tree edit distance represents the minimum cost of transforming one tree into another by a sequence of edit operations. There are three types of edit operations: inserting a new node, deleting an existing node, and renaming a node with a new label. Figure 1 provides a simple example, and the index $i$ of node $v_i$ corresponds to its position in the preorder traversal of the tree. In Figure 1, if the cost of each three edit operation is 1 then the TED between $\bar{X}$ and $\bar{Y}$ is 3. In real-world applications, the cost of edit operations usually varies for different nodes based on their labels and positions, and it can be user-defined as a cost matrix. The goal of TED computation is to find the minimum cost among all possible edit operation sequences.

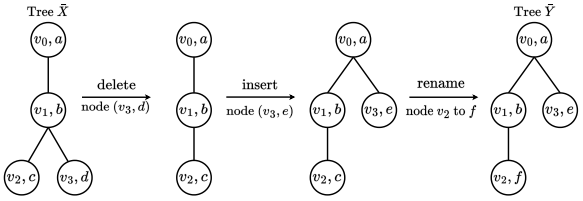



**Figure 1: Edit operations and edit distance**

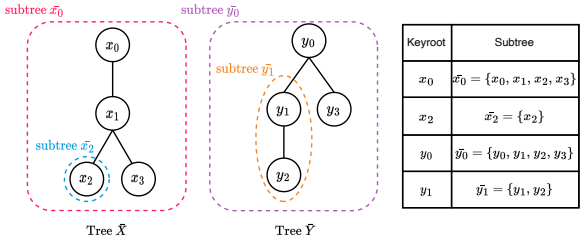
## 2.2 The Basic TED Algorithm

Dynamic programming is first introduced in [74] to address the TED problem for the basic and most widely-adopted TED algorithm, whose worst-case time complexity is $O(n^4)$. The algorithm consists of two primary steps. The first step involves identifying all keyroot nodes and the subtrees rooted at these keyroots, based on the tree structure. In the second step, for each pair of these subtrees, the algorithm utilizes a 2D table (matrix) to calculate the distance between subtrees by dynamic programming. Once all these tables have been computed, the algorithm returns the final distance between the two input trees.

*2.2.1 Keyroots and Subtrees.* In the preorder traversal manner [44], a node is identified as a keyroot if: (1) it is the root node, or (2) it has at least one right sibling. For the tree $\bar{X}$ in Figure 2, both $x_0$ and $x_2$ qualify as keyroots, whereas $x_1$ and $x_3$ are not keyroots.

The subtree rooted at a keyroot $x_i$ is denoted as tree $\bar{x_i}$. It includes the keyroot itself, and extends to the rightmost node it can reach which is denoted as $rl(x_i)$. The subtree $\bar{x_i}$ can hereby be represented as the set $\{x_i, x_{i+1}, \cdots, rl(x_i)\}$. For example, in Figure 2, $x_0$ is a keyroot of tree $\bar{X}$. The rightmost node that $x_0$ can reach is $x_3$; that is, $rl(x_0) = x_3$, and therefore the subtree $\bar{x_0}$ contains all nodes from $x_0$ to $x_3$ ($\bar{x_0} = \{x_0, x_1, x_2, x_3\}$). For another keyroot $x_2$, the rightmost node it can reach is itself ($rl(x_2) = x_2$), and thus the subtree $\bar{x_2}$

contains only one node, $x_2$. The keyroots and subtrees rooted at keyroots of $\bar{Y}$ are also shown in Figure 2.




| Keyroot | Subtree |
|---|---|
| $x_0$ | $\bar{x_0} = \{x_0, x_1, x_2, x_3\}$ |
| $x_2$ | $\bar{x_2} = \{x_2\}$ |
| $y_0$ | $\bar{y_0} = \{y_0, y_1, y_2, y_3\}$ |
| $y_1$ | $\bar{y_1} = \{y_1, y_2\}$ |

**Figure 2: Keyroots and subtrees rooted at keyroots**

It is worth noting that the subtree rooted at a keyroot can be a component of the whole tree (e.g., $\bar{y_1}$), or simply a leaf node (e.g., $\bar{x_2}$). Furthermore, if this keyroot node is the tree's root, its subtree is exactly the entire tree itself (e.g. $\bar{x_0} = \bar{X}$, $\bar{y_0} = \bar{Y}$).

*2.2.2 Tables in DP.* After identifying all keyroots and their corresponding subtrees from the two input trees, the basic TED algorithm then proceeds to the DP computation step. For each keyroot pair $(x_k, y_l)$, the algorithm utilizes a table to compute the distance between two subtrees ($\bar{x_k}$ and $\bar{y_l}$) rooted at these two keyroots.

In a bottom-up manner, the algorithm starts with the pair of keyroots that have higher preorder indices, and it then progressively advances towards the roots of two input trees. For example, as illustrated in Figure 2, there are four pairs in total that will be computed in order: $(\bar{x_2}, \bar{y_1})$, $(\bar{x_2}, \bar{y_0})$, $(\bar{x_0}, \bar{y_1})$, and $(\bar{x_0}, \bar{y_0})$. These four DP tables are displayed in Figure 3.

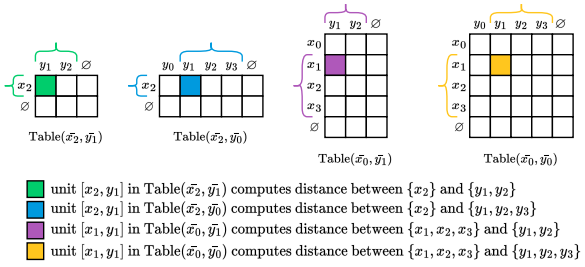



**Figure 3: DP Tables of each pair of subtrees in Figure 2**

We denote the table that computes the distance between subtree $\bar{x_k}$ and $\bar{y_l}$ as Table($\bar{x_k}, \bar{y_l}$). The row index of this table represents each $x$ node within the subtree $\bar{x_k}$, from $x_k$ to $rl(x_k)$, and an additional row is added at the end which represents an empty node ($\varnothing$). The column index stands for every $y$ node within subtree $\bar{y_l}$, from $y_l$ to $rl(y_l)$, and the last column is an empty node as well.

Inside a table, the unit with index $[x_i][y_j]$ computes the edit distance between a part of subtree $\bar{x_k}$ (from node $x_i$ to the end node $rl(x_k)$) and another part of subtree $\bar{y_l}$ (from node $y_j$ to the end node $rl(y_l)$). For example, since $rl(x_0) = x_3$ and $rl(y_1) = y_2$, the purple-colored unit $[x_1][y_1]$ in the Table($\bar{x_0}, \bar{y_1}$) indicates the distance between $\{x_1, x_2, x_3\}$ and $\{y_1, y_2\}$. Other unit examples are also shown in Figure 3.

*2.2.3 DP Recurrence.* In the basic TED algorithm, the DP recurrence formula serves as the rule for computing each unit in the tables. For any given Table($\bar{x_k}, \bar{y_l}$), we use $T_{i,j}$ to represent the unit with index $[x_i][y_j]$, which computes the edit distance between set $\{x_i, x_{i+1}, \cdots, rl(x_k)\}$ and $\{y_j, y_{j+1}, \cdots, rl(y_l)\}$ as in Figure 4. For the base case, the distance between two empty nodes $\varnothing$ is 0, and therefore $T_{\varnothing,\varnothing}$ is 0. The unit $T_{i,\varnothing}$ computes the distance between $\{x_i, x_{i+1}, \cdots, rl(x_k)\}$ and $\varnothing$, which is equal to the value of $T_{i+1,\varnothing}$ plus $cost(x_i, -)$, the cost of adding node $x_i$. Similarly, $T_{\varnothing,j}$ can be calculated by using $T_{\varnothing,j+1}$ plus $cost(-, y_j)$.

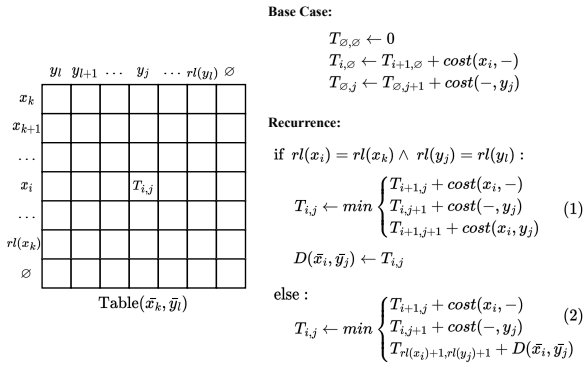



**Figure 4: DP Table and DP Recurrence Formula**

The value of $T_{i,j}$ relies on two different DP recurrence formulas. **Case 1:** If the rightmost node that can be reached by $x_i$ is the same as the one reached by $x_k$, meaning $rl(x_i) = rl(x_k)$, and simultaneously $rl(y_j) = rl(y_l)$, it implies that the set $\{x_i, x_{i+1}, \ldots, rl(x_k)\}$ forms the subtree $\bar{x_i}$ and $\{y_j, y_{j+1}, \ldots, rl(y_l)\}$ forms the subtree $\bar{y_j}$. In this scenario, $T_{i,j}$ can represent the minimal value among $T_{i+1,j} + cost(x_i, -)$, $T_{i,j+1} + cost(-, y_j)$, and $T_{i+1,j+1} + cost(x_i, y_j)$, where $cost(x_i, y_j)$ is the cost of renaming $x_i$ to $y_j$. In this case, $T_{i,j}$ is exactly the distance between $\bar{x_i}$ and $\bar{y_j}$, which will be stored and reused in subsequent computations. **Case 2:** Otherwise, $rl(x_i)$ is smaller than $rl(x_k)$ (in preorder manner), meaning subtree $\bar{x_i} = \{x_i, x_{i+1}, \ldots, rl(x_i)\} \subset \{x_i, x_{i+1}, \ldots, rl(x_k)\}$, or $rl(y_j)$ is smaller than $rl(y_l)$. Thus, in this second scenario, $T_{i,j}$ can be calculated as the minimal value among $T_{i+1,j} + cost(x_i, -)$, $T_{i,j+1} + cost(-, y_j)$, and $T_{rl(x_i)+1,rl(y_j)+1} + D(\bar{x_i}, \bar{y_j})$. The last term contains $D(\bar{x_i}, \bar{y_j})$ which is the edit distance between $\bar{x_i}$ and $\bar{y_j}$, computed in previous DP computations, plus the distance between the remaining part $\{rl(x_i) + 1, \ldots, rl(x_k)\}$ and $\{rl(y_j) + 1, \ldots, rl(y_l)\}$.

## 3 THE BOTTLENECKS IN TED COMPUTATION

In this section, we look into the dynamics of TED executions, examining existing DP models closely. We focus on parallel algorithms, identifying several critical issues and answering the question of why parallel processing for TED computation is so challenging.

### 3.1 Challenges in Parallel Processing of TED

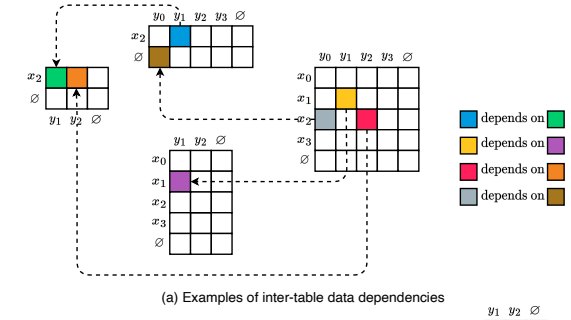
Due to the intrinsic high complexity, the transition from serial to massively parallel processing has become an inevitable trend in TED computing. However, parallelizing TED computation is an

exceedingly non-trivial task in both algorithm design and implementation for two primary reasons: (1) The algorithms and data structures of TED involve complex and extensive data dependencies, which significantly constraints parallelism and degrades parallel performance. (2) During TED computation, highly imbalanced distributions of workloads among tables are dynamically generated, preventing the development of a one-size-fits-all parallel algorithm.

These two issues are classical "headaches" in parallel processing [20, 65, 75, 76]. Moreover, for a series of subsequent optimized TED algorithms [15, 30, 46, 47], the DP framework remained, leading to the continuation of intricate data dependencies and an imbalanced distribution of workloads in these enhanced versions as well.

*3.1.1 Data Dependencies.* The DP recurrence Formula (1) and (2) in Figure 4 indicate that the computing process for tables in the basic algorithm exhibits inherent data dependencies, both among tables and within each individual table. For a unit $T_{i,j}$, its value might depend on either three of its neighbors as Formula (1), or two of its neighbors along with a computed result $D(\bar{x_i}, \bar{y_j})$ as Formula (2). It is important to highlight that $D(\bar{x_i}, \bar{y_j})$ is usually calculated from other proceeding tables, and hereby the computation of this table relies on the computations of other tables, leading to what we term as *inter-table dependency*. Additionally, the *intra-table dependencies* refer to the dependencies between $T_{i,j}$ and its neighbors.

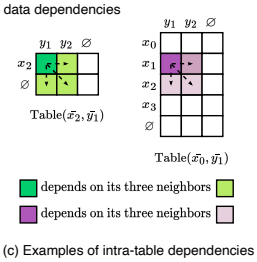
Figure 5 illustrates some data dependencies for the four tables in Figure 3. For example, the blue unit $[x_2][y_1]$ in Table$(\bar{x_2}, \bar{y_0})$, computes the distance between $\{x_2\}$ and $\{y_1,y_2,y_3\}$. According to the Formula (2), it relies on the result of $D(\bar{x_2}, \bar{y_1})$ plus the cost of inserting a node $y_3$, but the $D(\bar{x_2}, \bar{y_1})$ is computed by the green unit $[x_2][y_1]$ in Table$(\bar{x_2}, \bar{y_1})$. Therefore in the Figure 5(a), the blue unit depends on the green unit.




(a) Examples of inter-table data dependencies

| Table | Depends on | Num of inter-table Dependencies |
|---|---|---|
| $(\bar{x_2}, \bar{y_1})$ | *None* | 0 |
| $(\bar{x_2}, \bar{y_0})$ | $(\bar{x_2}, \bar{y_1})$ | 2 |
| $(\bar{x_0}, \bar{y_1})$ | $(\bar{x_2}, \bar{y_1})$ | 2 |
| $(\bar{x_0}, \bar{y_0})$ | $(\bar{x_2}, \bar{y_1}),(\bar{x_2}, \bar{y_0}),(\bar{x_0}, \bar{y_1})$ | 10 |

(b) Summary of inter-table dependencies




(c) Examples of intra-table dependencies

**Figure 5: Data dependency examples for tables in Figure 3**

When a unit within a table relies on certain units from other tables, it can be said that this table depends on those tables. Figure 5(b) presents the summary of all inter-table dependencies among these four tables. In this simple case of two four-node trees ($\bar{X}$ and $\bar{Y}$), the total number of inter-dependencies is 14. As the hierarchy of input trees grows larger, the inter-table dependencies become increasingly intricate and enormous. In addition, Figure 5(c) illustrates the intra-table data dependencies of two units, one from Table$(\bar{x_2}, \bar{y_1})$ and the other from Table$(\bar{x_0}, \bar{y_1})$, separately.

*3.1.2 Imbalanced workload.* As mentioned in Section 2.2.2 and Figure 4, for any given Table$(\bar{x_k}, \bar{y_l})$, if we denote the size of subtree $\bar{x_k}$ as $|\bar{x_k}|$ and the size of subtree $\bar{y_l}$ as $|\bar{y_l}|$, then the number of units that Table$(\bar{x_k}, \bar{y_l})$ needs to calculate is $(|\bar{x_k}|+1) \times (|\bar{y_l}|+1)$ in total. For two input trees that have thousands of nodes each, the size of the subtree rooted at each keyroot, $\bar{x_k}$ and $\bar{y_l}$, can range from one to thousands. Consequently, the dimensions of the tables might differ by a magnitude of millions, leading to highly imbalanced workloads during DP computations.

The example in Figure 5(a) shows that the sizes of the four tables are totally different from each other. Table$(\bar{x_0}, \bar{y_0})$ has the biggest size with 25 units overall, but the smallest one, Table$(\bar{x_2}, \bar{y_1})$ only has 6 units required to compute. On the other hand, these tables come in various shapes. Table$(\bar{x_2}, \bar{y_0})$ is horizontally broad while Table$(\bar{x_0}, \bar{y_1})$ is vertically elongated, and Table$(\bar{x_0}, \bar{y_0})$ is a square.

## 3.2 Issues in Existing Solutions

A parallel version of the basic TED algorithm is presented in [74], which offers a clear and comprehensive blueprint for exploiting the parallelism of the basic algorithm. Assuming each input tree comprises $n$ nodes, then the number of tables in the TED algorithm is $O(n^2)$ and each table has $O(n^2)$ units to compute. This parallel algorithm requires a substantial number of $O(n^3)$ processors, and thus each table has $O(n)$ processors to calculate all its entries. The parallel pattern is illustrated in Figure 6. Within a table, each anti-diagonal is treated as a wave so there are (2n-1) waves in total, from $w_1$ to $w_{2n-1}$. These $O(n)$ processors do parallel computing following the wavefront pattern from bottom-right to top-left, and units in the same wave across all tables can be processed concurrently. That is, all red units in $w_1$ from every table are computed in parallel first and subsequently all orange units from all tables are calculated simultaneously. This wave-by-wave procedure continues until all units in $w_{2n-1}$ have been computed.

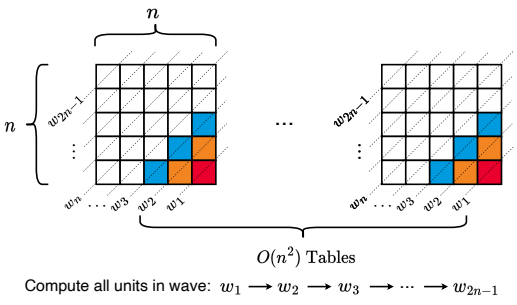



**Figure 6: Parallel version of the basic TED algorithm**

This framework works because for any given unit $T_{i,j}$, the neighbors it might rely on: $T_{i+1,j}$, $T_{i,j+1}$, and $T_{i+1,j+1}$, are all located in the preceding waves. Additionally, it has been proven in [74] that the value of $D(\bar{x_i}, \bar{y_j})$ is also computed in earlier waves (on other tables).

This ensures that all data dependencies are appropriately handled. However, implementing this algorithm in practice poses feasibility challenges due to poor utilization of computing and memory resources, as well as the high hardware cost, as explained below.

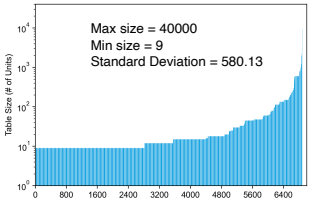
**Limitation (1): Huge memory space** is required. Although units in a table only depend on units on lower waves from itself or other tables, it does require all units from the nearest wave to be stored during computation, resulting in a substantial memory space requirement $O(n^3)$. As the dependencies across tables are not explicitly resolved, directly partitioning the tables into batches and computing each batch individually would disrupt the inter-table data dependencies, and thus, all tables must be processed together.

**Limitation (2): Frequent synchronization operations** are needed. For the computation of each wave, these processors need to synchronize before proceeding to the next wave. That is, the processors cannot compute units in $w_{i+1}$ unless all units in $w_i$ across all tables have been fully calculated. The frequent synchronization overhead among a vast number of processors is unacceptable.

**Limitation (3): Significant load-imbalance** persists for the majority of the execution time. Although the memory complexity for each table is $O(n^2)$, the actual table sizes range from minimal ones to $O(n^2)$. This disparity causes a significant load imbalance during computation. Yet, this parallel solution lacks a mechanism to allocate parallel resources suitably to tables with different sizes.

| | Execution Time (ms) |
|---|---|
| Zhang-Shasha's Parallel Solution on GPU | 485.24 |
| Basic TED on CPU | 15.61 |

(a) Performance Comparison




(b) Imbalanced table sizes (in log scale)

**Figure 7: Performance comparison between the parallel solution in [74] on GPU and the basic TED algorithm on CPU using 200-node trees.**

We have implemented this parallel TED algorithm on GPU and evaluated its performance on small trees with 200 nodes in Figure 7 (a). The execution time for this parallel version is nearly 32x longer than that of the sequential basic TED algorithm. Limited hardware resources and substantial synchronization cost severely degrade the overall performance of this parallel solution. Figure 7 (b) displays the range of table sizes in the test case; here, the table index merely serves as a label for referencing a table and has no specific meaning. The standard deviation in table sizes exceeds 580, highlighting a pronounced workload imbalance among them. An improved parallel algorithm is introduced in [73]. However, its implementation still requires a complex synchronization mechanism with significant overhead. Furthermore, the issue of imbalanced workload had also not been given sufficient attention.

## 3.3 Can we exploit unique dynamics of TED?

The three **inherent** limitations outlined in Section 3.2 are systemic, being persistent in the current framework, regardless of how we use other enhanced DP techniques to optimize the implementation. However, opportunities often reside within challenges.

It is essential to recognize that the DP computation in TED exhibits certain unique patterns in execution. The recurrence formulas in Figure 4 show that during the computation, the DP recurrence pattern continually changes, being directly influenced by the structure of the two input trees. Particularly in Formula (2), the location of $T_{rl(x_i)+1,rl(y_j)+1}$ is determined by the structures of subtrees $\bar{x_k}$ and $\bar{y_l}$. This paper addresses the following question:

*Can we design a new and efficient parallel framework for TED to address the three limitations by taking advantage of its unique dynamics in DP?*

We believe that it is only by leveraging the inherent dynamics in TED execution patterns that we can find the most efficient way to parallelize the TED computation. In contrast to the existing solutions, this paper aims to develop an effective parallel processing framework and its implementation on GPU with an insightful consideration of the unique execution dynamics of TED.

# 4 X-TED: AN OVERVIEW

## 4.1 Insights from Analysis

The bottleneck of the existing parallel solution arises because it cannot handle data dependencies efficiently. As discussed in Section 3.2, all units in a given wave $w_i$ across all tables must be calculated together. This means that for any table, if the units in $w_i$ have been computed, it cannot directly proceed to the wave $w_{i+1}$. Instead, it must wait for all units in $w_i$ from other tables to complete their computations, as its units in $w_{i+1}$ may depend on those units in $w_i$ from other tables according to the recurrence formula. This is the fundamental reason that causes Limitation (1) and Limitation (2).

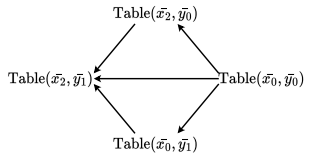
As a result, it is crucial to delve into the complex data dependencies among tables when designing a new parallel framework for TED computation. If we consider the computation of each table as a task, understanding the inter-table dependencies allows us to identify tasks that are independent of one another, and thus, can be processed concurrently. This helps to maximize the parallelism in practical implementations and eliminate the significant synchronization overheads across tables. Moreover, processors will require only $O(n^2)$ memory space to compute tables, which is a significant memory saving, compared to the previous solution's $O(n^3)$.

Additionally, the essence of DP recurrence in the basic TED algorithm implies that the serious load-imbalanced problem in Limitation (3) remains a persistent obstacle and impedes the parallel resource allocation. This must be addressed by efficient parallel programming to best utilize existing hardware resources in a performance- and cost-effective way.

## 4.2 Table Dependency Graph

A dependency graph is a graphical representation of the dependencies among tasks. Each node in the graph serves as a single task. If task $j$ depends on task $i$, a directed edge from node i to node j ($j \leftarrow i$) signifies this dependency. After treating the computation of each DP table as an individual task, we can construct

such a table dependency graph based on the inter-table dependency relationships between tables.




| Table | Depth of Table |
|---|---|
| ($\bar{x_2}, \bar{y_1}$) | 0 |
| ($\bar{x_2}, \bar{y_0}$) | 1 |
| ($\bar{x_0}, \bar{y_1}$) | 1 |
| ($\bar{x_0}, \bar{y_0}$) | 2 |

**Figure 8: Table dependency graph and depth of tables for the four tables in Figure 5**

For example, the dependency graph for the four tables from the Figure 5 is presented in Figure 8. The depth of a node in the dependency graph refers to the longest path from this node to the nodes without any dependency. It is evident that tables with the same depth value can be executed concurrently, as there is no dependency edge between them. In this example, it means that the Table($\bar{x_2}, \bar{y_0}$) and Table($\bar{x_0}, \bar{y_1}$) can be computed simultaneously.

## 4.3 Overview of Our Parallel Framework

In this paper, we present such a parallel framework named X-TED that is highly efficient and feasible for massive parallel TED computing. We first develop an innovative and efficient preprocessing algorithm that effectively resolves the inter-table data dependencies, based on the insights from the DP recurrence formula. And then we design a dynamic parallel strategy that allocates appropriate parallel resources to table with different sizes based on a mathematical model. The fundamental execution flow of our framework is illustrated in Figure 9.

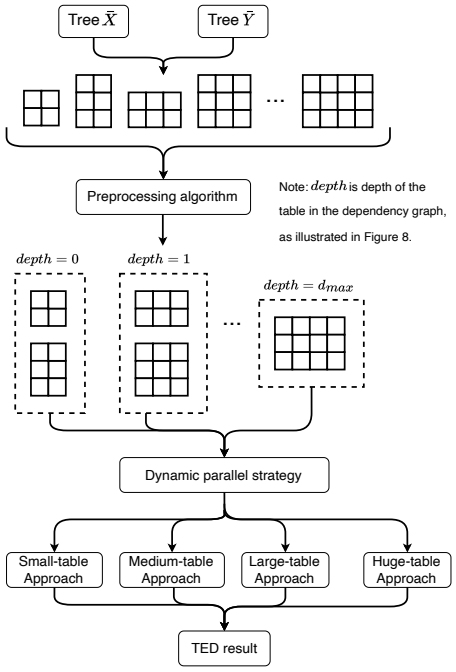



**Figure 9: Overview of X-TED framework**

Initially, the algorithm takes in two trees, $\bar{X}$ and $\bar{Y}$. Assuming $\bar{X}$ has $K$ keyroots and $\bar{Y}$ has $L$ keyroots, the number of DP tables required to compute is $(K \times L)$, and these tables has different sizes and shapes. Then, the efficient preprocessing algorithm we propose is employed to determine the depth of each table in the table dependency graph. **This is achieved solely by analyzing the tree structures, without the need to construct the table dependency graph**. Tables with identical depth can be processed concurrently, starting from depth = 0 and progressing to the maximum depth ($d_{max}$).

For the computation of tables at each depth, the dynamic parallel strategy we design adopts different parallel approaches to allocate hardware resources based on table sizes, which we categorize as small, medium, large, and huge. A mathematical model serves as a guide, determining the appropriate thresholds for each size type. This ensures that our parallel resource allocation strategies are applied accordingly and automatically during the parallel DP computation for TED. Details of the design, implementation, and performance results will be presented in the subsequent sections.

# 5 THE PREPROCESSING ALGORITHM

Data dependencies limit the concurrent processing of tasks and determine the execution sequence in parallel computing. In the TED algorithm, computing a single table can be viewed as a small task. When the input trees become larger and more complex, the dependencies among these tasks are increasingly convoluted. Therefore, before parallelizing the computing of massive tables in TED, it is vital to pinpoint these data dependencies and identify tasks that can run concurrently.

However, constructing the entire dependency graph and determining the depth of each table is both time-consuming and memory-intensive. In this section, we introduce an efficient preprocessing algorithm, which can calculate the depth of tables solely by analyzing tree structures. It simplifies dependency detection and groups independent tasks for later parallelization.

## 5.1 Keyroot Tree

Prior to introducing our preprocessing algorithm, we propose the concept of the *keyroot tree* of a given tree. The keyroot tree of a given tree $\bar{X}$ is an artificial tree built from $\bar{X}$ that keeps all keyroots of $\bar{X}$. The rules for establishing the keyroot tree are as follows. If a node $x_i$ is a non-keyroot node and a leaf, it will be removed directly. And then if $x_i$ is a non-keyroot node but not a leaf, it is replaced by edges which connect its parent and its children nodes. We denote the keyroot tree of $\bar{X}$ as $\overline{krX}$.

Figure 10 separately presents the keyroot trees of $\bar{X}$ and $\bar{Y}$ from Figure 2. The height of a tree is defined as the length of the longest path from that node to any leaf node it can reach. Thus, the height of node $x_0$ in $\overline{krX}$ is 1. And the height of $x_2$ should be 0 because it can only reach itself in $\overline{krX}$.

It is worth noting that compared to the original tree, the keyroot tree maintains the original hierarchy of all keyroots and only removes all non-keyroot nodes. In the keyroot tree $\overline{krX}$, if a keyroot $x_i$ can reach the node $x_j$ (from up to down), it means that in the original tree $\bar{X}$, the subtree rooted at $x_i$ includes the subtree rooted at $x_j$ ($\bar{x_i} \supseteq \bar{x_j}$). Otherwise there is no path from $x_i$ to $x_j$ in $\overline{krX}$.

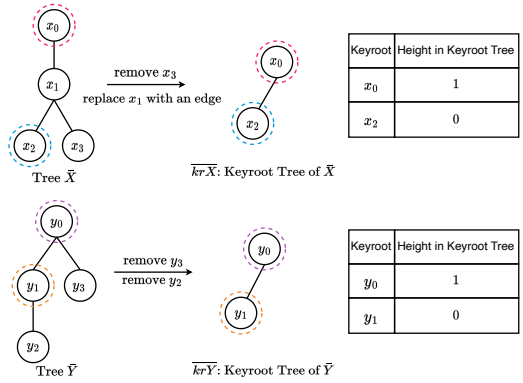



**Figure 10: Keyroot trees and depth of keyroots**

## 5.2 Essence of Inter-table Dependencies

According to the recurrence in Figure 4, for a given Table($\bar{x_k}, \bar{y_l}$), it depends on Table($\bar{x_p}, \bar{y_q}$) because some units within it rely on the results in Table($\bar{x_p}, \bar{y_q}$). From the perspective of tree structures, only if the subtree $\bar{x_k}$ contains the subtree $\bar{x_p}$ **or** subtree $\bar{y_l}$ contains the subtree $\bar{y_q}$, the processing of Table($\bar{x_k}, \bar{y_l}$) will rely on the computed results from Table($\bar{x_p}, \bar{y_q}$). This constitutes the fundamental reason behind the inter-table data dependencies, which means that the inherent hierarchical structures of two input trees determine the dependencies among tables in TED. Thus the *keyroot tree* we proposed in Section 5.1 serves to abstract the hierarchical structure of keyroots for input trees and aids in computing the depth of each table in the dependency graph. We propose our theorem as the basis of the preprocessing algorithm and prove it as below.

**Lemma 1.** *Assume a node in $\overline{krX}$ whose height is m denoted as $kx_m$, there must exist a sequence of keyroots $\{kx_0, kx_1, kx_2, ..., kx_{m-1}\}$ in the maximum path between $kx_m$ and a leaf node in $\overline{krX}$, and for node $kx_i \in \{kx_0, kx_1, kx_2, ..., kx_m\}$ the height of $kx_i$ in $\overline{krX}$ is i.*

**Theorem 1.** *Assume a node in $\overline{krX}$ whose height is m denoted as $kx_m$ and a node in $\overline{krY}$ whose height is n denoted as $ky_n$, the depth of Table($k\bar{x}_m, k\bar{y}_n$) in the table dependency graph is (m+n).*

$$Table\_Depth(k\bar{x}_m, k\bar{y}_n) = m + n \tag{3}$$

Proof. $kx_m$ is a node whose height is $m$ in $\overline{krX}$ and $ky_n$ is a node whose height is $n$ in $\overline{krY}$. According to Lemma 1, there exists a sequence of keyroots $\{kx_0, kx_1, kx_2, ..., kx_m\}$ and for node $kx_i \in \{kx_0, kx_1, kx_2, ..., kx_m\}$, the height of $kx_i$ in $\overline{krX}$ is i. Also, there is another sequence of keyroots $\{ky_0, ky_1, ky_2, ..., ky_n\}$ and for node $ky_j \in \{ky_0, ky_1, ky_2, ..., ky_n\}$, the height of $ky_j$ in $\overline{krY}$ is j.

We then use the introduction method to prove, and the base case is shown first. When $m = 0$ and $n = 0$, the keyroots are $kx_0$ and $ky_0$. Since the $kx_0$ and $ky_0$ are leaf nodes in keyroot trees, which means that in the original tree $\bar{X}$ and $\bar{Y}$, the subtrees correspondingly rooted at these two keyroots do not include any smaller keyroot subtree. This implies that the computing of Table($k\bar{x}_0, k\bar{y}_0$) does not depend on results of other tables in TED algorithm. Thus the depth of Table($k\bar{x}_0, k\bar{y}_0$) in the dependency graph is 0. That is,

$$\text{Table\_Depth}(k\bar{x}_0, k\bar{y}_0) = 0 \tag{4}$$

When $m = 0$ and $n = 1$, the two keyroot sequences are $\{kx_0\}$ and $\{ky_0, ky_1\}$. In $\overline{krY}$, node $ky_1$ and $ky_0$ are in the same path and there is only one edge between them because the height of $ky_1$ is larger than that of $ky_0$ by 1. This indicates that, the subtree $k\bar{y}_1$ in $\bar{Y}$ incorporates the subtree $k\bar{y}_0$, while the subtree $k\bar{x}_0$ in $\bar{X}$ does not encompass any smaller keyroot subtree. Further, the computing of Table($k\bar{x}_0, k\bar{y}_1$) relies on the result of Table($k\bar{x}_0, k\bar{y}_0$), so its depth in the dependency graph is 1. Similarly, when $m = 1$ and $n = 0$, the keyroots sequence are $\{kx_0, kx_1\}$ and $\{ky_0\}$. As the subtree $k\bar{x}_1$ contains the subtree $k\bar{x}_0$ in $\bar{X}$, the Table($k\bar{x}_1, k\bar{y}_0$) depends on the data in Table($k\bar{x}_0, k\bar{y}_0$) as well.

$$\text{Table\_Depth}(k\bar{x}_0, k\bar{y}_1) = 1 \tag{5}$$

$$\text{Table\_Depth}(k\bar{x}_1, k\bar{y}_0) = 1 \tag{6}$$

Thus the statement is true for the base case. Then, in the introduction procedure, we assume the formula below is true.

$$\text{Table\_Depth}(k\bar{x}_{m-1}, k\bar{y}_{n-1}) = (m-1) + (n-1) \tag{7}$$

In $\overline{krX}$, node $kx_m$ is higher than $kx_{m-1}$ by 1 which reveals that in the input tree $\bar{X}$, the subtree $k\bar{x}_{m-1}$ is contained by $k\bar{x}_m$. As a result, there should be an edge from the Table($k\bar{x}_m, k\bar{y}_{n-1}$) to the Table($k\bar{x}_{m-1}, k\bar{y}_{n-1}$) because the former depends on the results of latter. The depth of Table($k\bar{x}_m, k\bar{y}_{n-1}$) in the graph should be the depth of Table($k\bar{x}_{m-1}, k\bar{y}_{n-1}$) plus 1. Similarly, the depth of Table($k\bar{x}_{m-1}, k\bar{y}_n$) also equals that of Table($k\bar{x}_{m-1}, k\bar{y}_{n-1}$) plus 1.

$$\text{Table\_Depth}(k\bar{x}_m, k\bar{y}_{n-1}) = (m-1) + (n-1) + 1 = m + n - 1 \tag{8}$$

$$\text{Table\_Depth}(k\bar{x}_{m-1}, k\bar{y}_n) = (m-1) + (n-1) + 1 = m + n - 1 \tag{9}$$

To figure out the depth of Table($k\bar{x}_m, k\bar{y}_n$), it is easy to notice that the subtree $k\bar{x}_m$ includes the subtree $k\bar{x}_{m-1}$ and the subtree $k\bar{y}_n$ contains the subtree $k\bar{y}_{n-1}$. Consequently, the Table($k\bar{x}_m, k\bar{y}_n$) depends on both Table($k\bar{x}_m, k\bar{y}_{n-1}$) and Table($k\bar{x}_{m-1}, k\bar{y}_n$). The depth of Table($k\bar{x}_m, k\bar{y}_n$) should be higher than their depths by 1. Hence, the final depth is:

$$\text{Table\_Depth}(k\bar{x}_m, k\bar{y}_n) = m + n - 1 + 1 = m + n \tag{10}$$

Formula (10) is the same as Formula (3). Both the base case and inductive cases have been successfully proven. Thus, Theorem 1 holds true for all natural numbers for $m$ and $n$.

□

According to Theorem 1, it means that the depth of Table($\bar{x_i}, \bar{y_j}$) in the dependency graph, is equivalent to the sum of the height of node $x_i$ in $\overline{krX}$ and the height of the node $y_j$ in $\overline{krY}$. In Figure 10, the height of $x_2$ in $\overline{krX}$ is 0 and the height of $y_1$ is 0. Therefore, the depth of Table($\bar{x_2}, \bar{y_1}$) is 0. Similarly, both depths of Table($\bar{x_2}, \bar{y_0}$) and Table($\bar{x_0}, \bar{y_1}$) are: 0 + 1 = 1, and the depth of Table($\bar{x_0}, \bar{y_0}$) is 2. The results are exactly the same as the depth of these tables in the dependency graph in Figure 8.

## 5.3 The Preprocessing Algorithm

In the table dependency graph, tables that have identical depths are independent with each other, which provides an opportunity to perform parallel processing. Therefore, we propose an algorithm which can efficiently determine the depth of tables in the dependency graph without building the entire graph by using knowledge from Section 5.2. The detailed algorithm is shown in Algorithm 1.

**Algorithm 1** Dependency preprocessing: computes depths of all tables

**Input:** Two input trees $\bar{X}$ and $\bar{Y}$
**Input:** Two keyroots array $kr_{\bar{X}}$ and $kr_{\bar{Y}}$ ▷ All keyroots of $\bar{X}$ and $\bar{Y}$
**Output:** Heights of all keyroots in two keyroot trees ($kh_{\bar{X}}$ and $kh_{\bar{Y}}$)
**Output:** Depths of all tables ($td$)

**procedure** TABLE-DEPTH
  $kh_{\bar{X}} \leftarrow$ KEYROOT-HEIGHT($kr_{\bar{X}}$)
  $kh_{\bar{Y}} \leftarrow$ KEYROOT-HEIGHT($kr_{\bar{Y}}$)
  **for** $k \leftarrow kr_{\bar{X}}.size()$ **downto** 1 **do**
    **for** $l \leftarrow kr_{\bar{Y}}.size()$ **downto** 1 **do**
      ▷ depth of table is the sum of heights of two keyroots
      $td[k][l] \leftarrow kh_{\bar{X}}[k] + kh_{\bar{Y}}[l]$
    **end for**
  **end for**
**end procedure**

**procedure** KEYROOT-HEIGHT
  **for** $i \leftarrow kr.size()$ **downto** 1 **do**
    **if** $kr(i)$ is a leaf node $\vee$ $kr(i)$ has only **one** non-keyroot child **then**
      ▷ $kr(i)$ is a leaf node in the keyroot tree
      $kh[i] \leftarrow 0$
    **else**
      **for** $j \leftarrow kr.size()$ **downto** $i$ **do**
        **if** $kr(j) \subseteq kr(i)$ **then**
          ▷ subtree $kr(i)$ includes subtree $kr(j)$
          $kh[i] \leftarrow max(kh[i],\ kh[j] + 1)$
        **end if**
      **end for**
    **end if**
  **end for**
**end procedure**

As previously mentioned, the depth of the table is equal to the sum of the heights of two keyroots in the corresponding keyroot tree. The Algorithm 1 consists of two main steps. In the first step, the heights of all keyroots in the keyroot tree are calculated. After computing the heights of all keyroots for tree $\bar{X}$ and $\bar{Y}$, the depth of table that computes the distance between any two subtrees can be directly computed by the addition of two heights.

The running time complexity of our preprocessing algorithm is $O(n^2)$, considering that the number of nodes for each two input trees is $n$. In the step of computing heights, it iterates over all keyroots and checks whether the keyroots with larger indices are included in the subtree rooted at a particular keyroot. Since the preorder manner is applied, only keyroots with larger indices are possible to be contained. The complexity of this procedure is $O(n^2)$. In the step of calculating table depths, the algorithm traverses all keyroots in the two input trees and performs an addition. Thus, the running time complexity of this step is also $O(n^2)$.

Compared to the cubic-level complexity of TED algorithms, this preprocessing algorithm is efficient and can directly compute the depths of all tables without constructing a complete dependency graph. It provides a clear guide for the parallelization of the basic TED algorithm and a series of later optimized solutions.

# 6 DYNAMIC PARALLEL STRATEGIES

## 6.1 Execution Patterns in Processing

By employing the preprocessing algorithm, we can rapidly calculate the depths of all tables in the table dependency graph. Subsequently, tables in the same depth can be processed in parallel. This method ensures a systematic and parallelized processing of tables in the TED algorithm, removing the need to analyze the complex relationships between different tables.

Parallel computing starts with all tables that have a depth of 0. We utilize a task list to store these tables, and the processors fetch tasks from this list for computation. After all tables at the current depth have been computed, it proceeds to the next depth level and calculates all tables at that depth concurrently. The computation finishes once the last table at the largest depth ($d_{max}$) is calculated.

Yet, the results obtained by only using this simple framework are far from being satisfactory. We implemented this parallel solution on GPU in a straightforward manner, where each thread is assigned to compute one table and threads that handle tables with identical depths run concurrently. In a small test case involving two 200-node trees, the basic sequential algorithm took 15.61 milliseconds while this naive parallel version on GPU required 13.78 milliseconds.

The primary performance bottleneck in this naive implementation arises from the severe load imbalance during parallel computation. Processors calculating small tables are underutilized, while those dealing with large tables are overloaded. This imbalance results in substantial idle time and wastage of valuable computing resources. Figure 11 presents the size distribution of all tables within each depth for this test case. Additionally, the last plot illustrates both the standard deviation of all table sizes and the difference between the largest and smallest size at each depth in the test case.

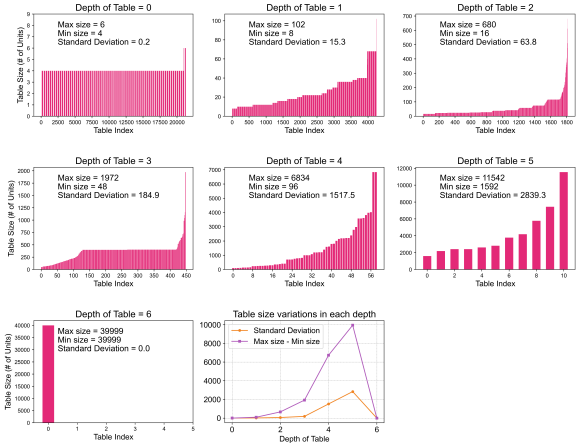



**Figure 11: Variations of table sizes by depth in the test case**

Three distinct parallel stages can be apparently identified based on the variation in table sizes across all depths. The first stage is the embarrassing processing, where tables at small depths have fairly balanced size and each table can be calculated by a single processing unit. In the second stage, as depth increases, table sizes become significantly large, and the standard deviation also increases, intensifying the issue of workload imbalance, which is termed imbalanced processing. Finally, at the maximum depth, as there is only one but the biggest table (which depends on all other tables), all parallel resources thus should be dedicated to compute it, which is the focused processing stage. Therefore, a dynamic parallel strategy should be built to resolve these complicated workload changes during parallel TED computation.

## 6.2 Dynamic Parallel Strategy

*6.2.1 Different Approaches.* As discussed above, tables with different sizes should be processed using different parallel approaches.

This guarantees that all processing units are utilized efficiently, and the load imbalanced issue can be effectively addressed.

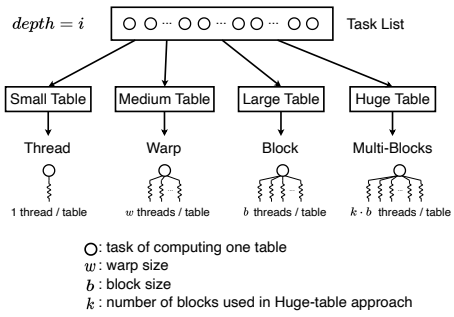



**Figure 12: Dynamic parallel strategy on GPU**

Figure 12 illustrates the dynamic strategy we designed on GPU that are utilized for computing DP tables in the TED problem. All tables with a specified depth $i$ are in a task list and will be categorized into four size types: small, medium, large, and huge, based on their sizes. Different approaches (kernel functions on GPU) are subsequently applied to each category.

The first approach is single-thread computing for small tables, where all units in the table are computed by a single thread. The second approach employs the single-warp computing for medium-sized tables. In this method, each thread in the warp computes entries in a single row, and all units in the same wave are processed simultaneously, following the wavefront parallel pattern. After each wave's computation is completed, all threads within the warp synchronize. When dealing with tables containing numerous rows, the warp starts from the bottom and iterates upwards to the top.

The third method is termed single-block computing, which follows a similar execution pattern to previous methods, but the difference is that this method requires block-level synchronization at the end of each wave. The last one is multi-blocks computing, where threads from multiple $k$ blocks collaborate to process a single table and inter-block synchronizations are performed.

*6.2.2 Approach Switch.* The size of table serves as a metric that determines the optimal approach for table computing among the four parallel methods mentioned above. The approach switch pattern is illustrated in Figure 13. Tables that have small entries can be directly processed by using single-thread approach, where each table is calculated by one single thread. As the table size grows to a medium scale, it is more efficient to use the single-warp method. Therefore, there exists a threshold $\theta_1$ in table size that can be utilized to choose from these two methods.

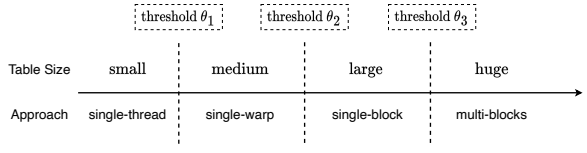



**Figure 13: Parallel approach changes with table size**

Similarly, for tables with large units, the single-block approach should be used to exploit more concurrent processing units. The threshold $\theta_2$ represents this change point. When the table grows to huge-level size, the multi-blocks computing should be employed although it introduces the overhead of synchronizations between multiple blocks. The threshold here is marked as $\theta_3$. The values for these three thresholds can be determined using the mathematical model described in Section 6.3.

## 6.3 Mathematics Model for Approach Switch

As previously mentioned, X-TED relies on three crucial thresholds to determine the approaches applied in DP table computing. Hence, accurately finding the values of these thresholds is of utmost importance as it directly impacts our parallel framework's performance. We introduce a mathematical model that provides guidance in identifying the ideal values for these thresholds, ensuring that X-TED achieves optimal performance.

Assume the computing time for each unit in a table is identical and denoted as $t_u$, the warp size is $w$, the block size is $b$, the number of blocks used in the multi-blocks method is $k$. For a table with $n$ rows and $n$ columns, its has (2n-1) waves in total, and the computation time using the four methods can be calculated as follows. The inter-block synchronization for $k$ blocks is marked as $sync_m(k)$.

$$T_{thread} = n^2 \cdot t_u \tag{11}$$

$$T_{warp} = \begin{cases} (2n-1) \cdot (t_u + sync_{warp}) & n \le w \\ \lceil \frac{n}{w} \rceil \cdot (w + n - 1) \cdot (t_u + sync_{warp}) & n > w \end{cases} \tag{12}$$

$$T_{block} = \begin{cases} (2n-1) \cdot (t_u + sync_{block}) & n \le b \\ \lceil \frac{n}{b} \rceil \cdot (b + n - 1) \cdot (t_u + sync_{block}) & n > b \end{cases} \tag{13}$$

$$T_{mblocks} = \begin{cases} (2n-1)(t_u + sync_m(k)) & n \le k \cdot b \\ \lceil \frac{n}{k \cdot b} \rceil (k \cdot b + n - 1)(t_u + sync_m(k)) & n > k \cdot b \end{cases} \tag{14}$$

As the table size changes, these four formulas generate four distinct curves, representing the time taken for each parallel approach. The threshold $\theta_1$, $\theta_2$, and $\theta_3$ represent the intersection points between Formula (11) and Formula (12), Formula (12) and Formula (13), Formula (13) and Formula (14), respectively.

From formulas above, four machine-dependent variables ($t_{unit}$, $sync_{warp}$, $sync_{block}$, and $sync_m(k)$) determine the curves and their intersection points. For a given GPU, once the values of these four variables are obtained, calculating the three intersection points and determining the thresholds becomes straightforward. These thresholds serve as criteria to dynamically select different approaches within X-TED and can be directly applied in all types of substantial TED computations running on this machine, which facilitates the efficient utilization of the GPU resources.

# 7 PERFORMANCE EVALUATION

## 7.1 Evaluation Platform Settings

*7.1.1 Baseline.* We have implemented the basic TED algorithm [74] and the state-of-the-art and optimized TED algorithm, AP-TED [47], as sequential baselines of CPU-based algorithms. To fairly compare the parallel performance, we have also deployed another online CPU solution called Multi-Core TED (MC-TED) [3], as it is the state-of-the-art parallel TED implementation. Additionally, we

implemented a highly optimized parallel CPU-version of X-TED to compare the performance of X-TED on the GPU.

*7.1.2 Experimental Environment.* All implementations are evaluated on a workstation with Intel Core i9-12900 CPU, 64 GB memory, and NVIDIA RTX 3090 (24GB). Both MC-TED and the parallel-CPU version of X-TED use 8 cores in evaluation. We also deploy X-TED on two cloud-available GPUs: NVIDIA A100-SXM4 (40 GB) and H100-PCIe (80 GB). All code is compiled using *O*3 optimizations. The cmake version is 3.25.3 and CUDA is 12.2.

*7.1.3 Datasets.* We evaluated all implementations on real-world datasets from four distinct domains, including various types of trees. The characteristics of the trees in these datasets are detailed in Table 1. "Avg. Nodes" indicates the average number of nodes per tree for each dataset, while "Max. Nodes" refers to the node count of the largest tree in each dataset.

Swissport [11] contains XML files for protein sequences and descriptions, which is one of the most widely used protein databases around the world, consisting of large-sized and flat trees. Python [50] dataset has recently become popular for training deep learning models in AI programming [26, 32, 63, 69], which contains large-sized and high JSON parse trees of python programs. DBLP dataset [14] comprises millions of small-sized XML bibliographic records from the dblp computer science bibliography database. The last dataset is the Bolzano street dataset [4], which includes street addresses of Bolzano city in medium-sized hierarchical tree structures.

**Table 1: Characteristics of trees in each dataset**

| Dataset | Max. Depth | Avg. Depth | Avg. Nodes | Max. Nodes |
|---|---|---|---|---|
| Swissport | 9 | 7.01 | 988.36 | 7241 |
| Python | 156 | 13.11 | 927.41 | 8516 |
| DBLP | 7 | 3.16 | 26.05 | 1186 |
| Bolzano | 4 | 3.82 | 178.71 | 2105 |

*7.1.4 Evaluation Method.* In each dataset, we select trees at appropriate intervals, using rational size increments, where the size of trees ranges from small (100 nodes) to large (1000 nodes). For a data point corresponding to tree size $i$, we randomly choose 5 pairs of trees with node counts near $i$, ensuring the average node count for trees in each pair is $i$. The computation time for calculating TED between these tree pairs is measured and averaged in 20 runs.

## 7.2 Overall Performance

The overall performance of all five implementations is presented in Figure 14. The performance of our X-TED framework on GPU far surpasses all other solutions. As the tree size increases, the computation times for both the basic TED algorithm and AP-TED, increase drastically, due to the intrinsic high-complexity of the TED algorithms. For the MC-TED implementation, although it employs parallel techniques to accelerate computing, the performance is far from satisfactory. In some cases of DBLP dataset, its processing time is even worse than that of AP-TED. This is mainly because in MC-TED, the computing order of tables strictly follows the table dependency graph, resulting in high synchronization overhead. And the severe load imbalanced problem among tables is not paid attention and remains unaddressed. Compared to the AP-TED algorithm, our CPU-version of X-TED can achieve an impressive speedup of up to 6.5x. Further, in contrast of the performance of this CPU-version X-TED , the X-TED implementation on GPU achieves even more remarkable speedup of up to 10.9x in computation time. The X-TED framework on GPU outperforms the state-of-the-art sequential algorithm (AP-TED) by an average speedup of 42x.

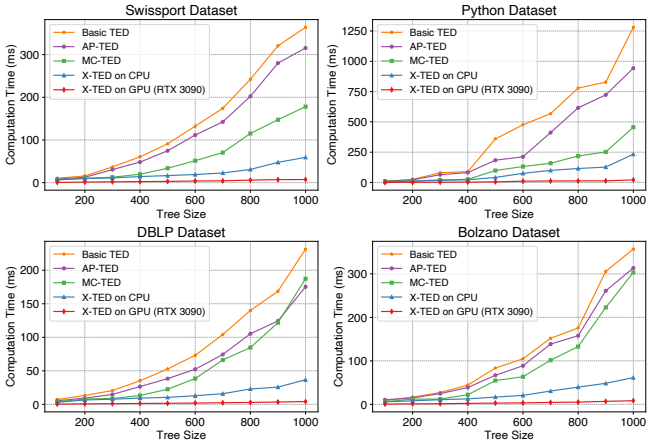



**Figure 14: Computation time for TED between trees with different sizes on 4 datasets**

## 7.3 High Speedup and Efficient Preprocessing

Figure 15 presents a more detailed comparison of all five implementations on tree pairs with 1000 nodes. For the X-TED running on CPU, its speedup compared to the basic TED algorithm ranges from 5.48x to 6.3x. Moreover, when compared to AP-TED, it achieves an average speedup of 4.8x. In contrast to another multi-core implementation, MC-TED, it achieves an average speedup of 3.8x. This demonstrates the advantage and effectiveness of using the preprocessing algorithm to resolve the inter-table dependency issue and results in enhanced parallelism.

When compared with the basic TED algorithm, the CPU implementation of X-TED exhibits a higher speedup on DBLP dataset compared to Python dataset. This is because trees in Python dataset have greater depth than those in DBLP dataset. The hierarchy of trees directly influences the depth of tables in the TED computation. Shallower trees result in less deep tables, allowing most DP tables to reside in the initial depth levels, facilitating efficient parallel computation for multi-core CPUs.

For the X-TED framework on GPU, its speedup impressively achieves up to 60x speedup compared to the basic TED algorithm and up to 44x speedup compared to the AP-TED. Also, it surpasses the performance of MC-TED by an average speedup of 31x. In addition, it outperforms the CPU-version X-TED by a speedup of up to 11x and an average of 8.8x, because of the benefits of employing flexible dynamic parallel strategies in GPU implementation which addressed the load imbalanced problem effectively.

The time consumed by the our preprocessing algorithm is also included in the measurement. The preprocessing time for both CPU-version X-TED and GPU-version X-TED is identical as it is

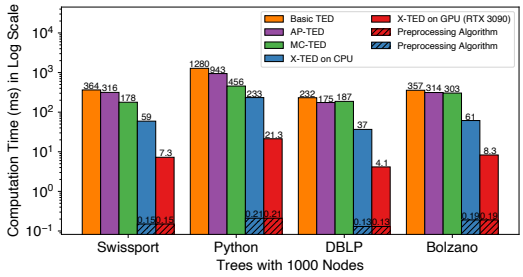



**Figure 15: Performance comparison (in log scale) for 1000-node trees within 4 datasets**

executed by CPU. For the X-TED framework on GPU, the preprocessing time ranges from 0.13 to 0.21 microseconds, constituting approximately averaged 2% of the total computation time. This indicates the efficiency of the preprocessing algorithm as it has relatively low complexity compared to the TED algorithms.

## 7.4 Effectiveness and Necessity

To demonstrate the effectiveness of our design, we compared the performance of three versions: (1) the basic TED algorithm, (2) the TED algorithm with preprocessing and an naive parallel implementation, and (3) the TED algorithm with both preprocessing and dynamic parallel implementation. We executed all test trees ranging from 100 nodes to 1000 nodes in each dataset and recorded their total running time. The results are presented in Figure 16.

It is manifest that using the preprocessing algorithm to exploit the parallelism of table computation has improved the total running time. However, it fell short of achieving a satisfactory speedup, with a maximum speedup of only 1.76x. This is directly attributed to the serious load imbalanced problem we mentioned before. After employing the dynamic parallel strategy, the speedup has been impressively enhanced, reaching up to 30x compared to the version with only preprocessing and naive parallel method.

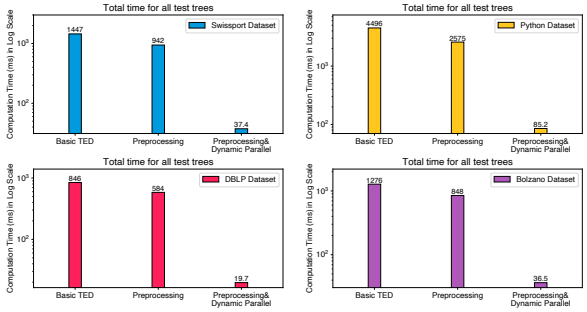



**Figure 16: Effectiveness of the preprocessing algorithm and dynamic parallel strategy**

In the dynamic parallel strategy design for GPU, four parallel approaches have been applied and their effectiveness has been assessed in Figure 17. It is evident that the performance is undesirable when only using the single-thread approach, as some threads end up working on a large table while some remain idle after completing

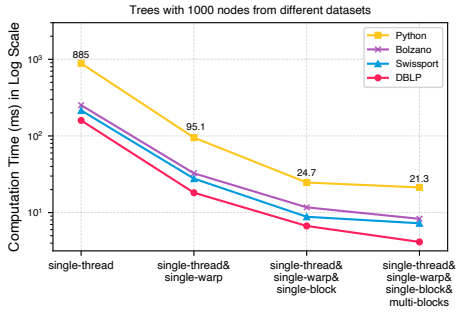



**Figure 17: Effectiveness of four parallel approaches in the dynamic parallel strategy**

few units. When the single-warp approach is introduced, it achieves a speedup of 9.3x compared to the first version. Based on this, a further speedup of 3.85x has been obtained by adding the single-block method because it can compute large tables more efficiently than the single-warp. Finally, the multi-block method improves the running time by 15%, which achieves the best performance.

The decrease in speedup comes from the following reason. The single-block and multi-block approaches are well-suited for large and huge tables, but these tables are in the minority and therefore the speedup is not so notable. Although there are a small number of such large tables, the single-thread and single-warp methods cannot be used as each thread will be overloaded. Therefore, it is necessary to incorporate all four approaches into the dynamic parallel strategy to achieve optimal workload allocation on GPUs.

## 7.5 Memory Usage

In Section 3.2 and 4.1, it is revealed that one significant limitation of the existing parallel solution in [74] is that it requires a substantial amount of memory space. In contrast, X-TED has resolved inter-table data dependencies, thus reducing the memory requirement to $O(n^2)$. We implement the existing parallel TED algorithm on GPU, and compare memory usage with that of X-TED in Figure 18.

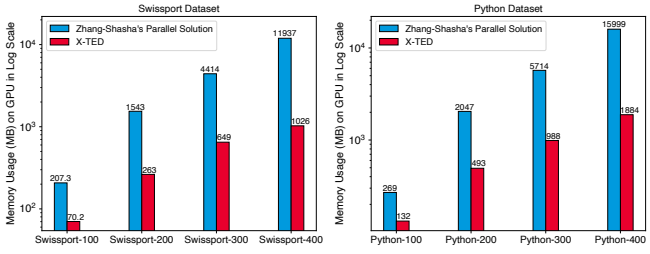



**Figure 18: Memory usage comparison on GPU between the parallel solution in [74] and X-TED across two datasets**

The memory usage for computing the TED between pairs of trees with 100, 200, 300, and 400 nodes respectively, is recorded using Swissport dataset and Python dataset. As the input tree size increases, X-TED achieves up to 11.6x memory space saving compared to the existing parallel algorithm. Furthermore, when the input trees have more than 500 nodes, the memory space required

by the existing parallel solution exceeds the capacity of RTX 3090, making it infeasible to compute edit distances between larger trees.

## 7.6 Scalability Evaluation

To further evaluate the scalability of X-TED, we deploy it on cloud machines from Lambda company [31], and test its performance on two other types of NVIDIA GPUs: A100-SXM4 (40 GB) and H100-PCIe (80 GB). In the experiment, we choose pairs of extra-large trees, with node counts ranging from 1000 to 6000, from Python dataset. Additionally, we generate random recursive trees [12] with a range of 1000 to 9000 nodes as synthetic trees, and evaluate X-TED on this Synthetic dataset as well, as shown in Figure 19.

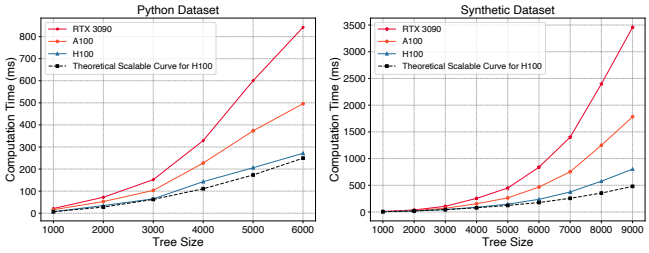



**Figure 19: Computation time between pairs of extra-large trees for X-TED on different GPUs**

Compared to RTX 3090, the speedup of A100 ranges from 1.42x to 1.94x. Moreover, H100 outperforms RTX 3090 by a speedup of up to 4.32x and surpasses the performance of A100 by up to 2.23x. The black curve in Figure 19 is the theoretical scalable performance curve for H100. For instance, H100 consumes 5.93 milliseconds to compute the TED between two 1000-node trees on the Synthetic dataset. Theoretically, when the size of both input trees is doubled to 2000 nodes, the computation time is expected to quadruple. Therefore, we plot such a curve as a reference. The small discrepancy between the actual curve of H100 and its theoretical scalable curve highlights the notable scalability of X-TED.

## 7.7 Performance under Memory Constraints

When dealing with extra-large trees that have thousands of nodes, the memory space needed for a single table becomes substantial. The memory capacity of GPU directly determines the number of such large tables that can be processed concurrently and thus limits the processing performance of X-TED. Therefore, we test the performance of X-TED on H100 under different memory constraints.

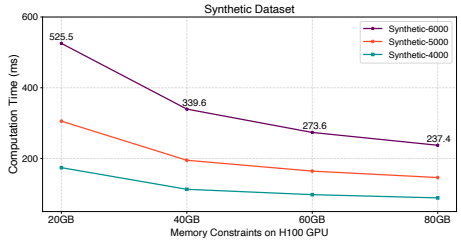



**Figure 20: Computation time of X-TED on H100 under different memory constraints**

The three lines in Figure 20 represent the computation time of X-TED for pairs of synthetic trees with 6000, 5000, and 4000 nodes, respectively. When the memory constraint on H100 is increased from 20 GB to 40 GB, the TED computation for trees with 6000 nodes achieves a speedup of 1.54x. Further, increasing the memory from 40 GB to 80 GB results in an additional speedup of 1.43x. The speedup gained from expanding memory space diminishes because once the memory on GPU is sufficient to store all tables at a given depth, further increasing the memory will not result in more tables being processed in parallel, and therefore will not contribute to further acceleration.

# 8 RELATED WORK

The first TED algorithm with complexity of $O(n^6)$ is proposed by Tai [59]. After [74], several optimized algorithms [15, 30, 46–48, 53] have improved the worst-case complexity to $O(n^3)$, which is the theoretical lower bound [19]. Some recent work [13, 42, 54] aimed to break the cubic complexity on specific constrains.

There has been scant research [73, 74] on developing parallel TED algorithms. In 2015, Shukla et al. [55] made an attempt. However, only tables that record the distances between lowest subtrees were computed in parallel. All other substantial tables were processed sequentially on CPU. The imbalanced workload problem on GPU was ignored as well. Another existing parallel work is an online project, MC-TED [3], that we tested in the evaluation.

Many prior studies e.g. [1, 5, 21, 38, 40, 61] have explored parallel techniques for general DP problems. Primarily, most related research e.g. [17, 25, 36, 57, 62, 66] focused on optimizing techniques under the wavefront parallel framework. Some studies [16, 35, 41, 60] aimed at enhancing the data locality, and some concentrated on refining synchronization mechanisms [6, 33, 39, 58, 64, 67, 68, 71].

# 9 CONCLUSION

We have developed X-TED, a high-performance massive parallel computation framework for TED computation and its implementation on GPU. We have identified two critical issues of TED: the complex and extensive data dependencies, and highly imbalanced workloads during parallel processing. These two issues have not been fundamentally addressed since the basic TED algorithm was published in 1989. Our preprocessing algorithm can quickly determine data dependency relationships among tables before parallel processing. In addition, we have applied effective parallel methods to compute tables with different sizes, optimizing resource allocation and effectively resolving the imbalanced workload issue. The performance result of X-TED remarkably outperforms all existing TED solutions. The superiority of X-TED performance shows its high effectiveness of massive parallel processing for TED. We believe the X-TED framework, along with the accompanying open-source software, fundamentally address the structural issues of parallel processing for TED.

# ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. The work is supported in part by the U.S. National Science Foundation under grants MRI-2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC-2310510.

# REFERENCES


[1] AmirAli Abdolrashidi, Devashree Tripathy, Mehmet Esat Belviranli, Laxmi Narayan Bhuyan, and Daniel Wong. 2017. Wireframe: Supporting Data-Dependent Parallelism through Dependency Graph Execution in GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 600–611. https://doi.org/10.1145/3123939.3123976

[2] Tatsuya Akutsu. 2010. Tree edit distance problems: Algorithms and applications to bioinformatics. *IEICE transactions on information and systems* 93, 2 (2010), 208–218.

[3] Alireza S. Abyaneh. 2020. Multi Core Tree Edit Distance. https://github.com/aabyaneh/MCTED Accessed on July 1, 2020.

[4] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. 2008. The Pq-Gram Distance between Ordered Labeled Trees. 35, 1, Article 4 (feb 2008), 36 pages. https://doi.org/10.1145/1670243.1670247

[5] Mehmet E. Belviranli, Chih-Hsun Chou, Laxmi N. Bhuyan, and Rajiv Gupta. 2014. A Paradigm Shift in GP-GPU Computing: Task Based Execution of Applications with Dynamic Data Dependencies. In *Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing* (Vancouver, BC, Canada) *(DIDC '14)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/2608020.2608024

[6] Mehmet E. Belviranli, Peng Deng, Laxmi N. Bhuyan, Rajiv Gupta, and Qi Zhu. 2015. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/2751205.2751243

[7] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1 (2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[8] Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Tree Edit Distance Cannot Be Computed in Strongly Subcubic Time (Unless APSP Can). *ACM Trans. Algorithms* 16, 4, Article 48 (jul 2020), 22 pages. https://doi.org/10.1145/3381878

[9] Weimin Chen. 2001. New Algorithm for Ordered Tree-to-Tree Correction Problem. 40, 2 (aug 2001), 135–158. https://doi.org/10.1006/jagm.2001.1170

[10] Jimmy Ka Ho Chiu and Yi-Ping Phoebe Chen. 2017. A comprehensive study of RNA secondary structure alignment algorithms. *Briefings in Bioinformatics* 18, 2 (2017), 291–305.

[11] The UniProt Consortium. 2022. UniProt: the Universal Protein Knowledgebase in 2023. *Nucleic Acids Research* 51, D1 (11 2022), D523–D531. https://doi.org/10.1093/nar/gkac1052

[12] Wikipedia contributors. 2024. Random recursive tree — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Random_recursive_tree [Online; accessed 14-January-2024].

[13] Debarati Das, Jacob Gilbert, MohammadTaghi Hajiaghayi, Tomasz Kociumaka, Barna Saha, and Hamed Saleh. 2022. Õ(n+poly(k))-time Algorithm for Bounded Tree Edit Distance. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 686–697. https://doi.org/10.1109/FOCS54457.2022.00071

[14] The dblp team. 2023. The dblp computer science bibliography. Monthly snapshot release of June 2023. https://dblp.org/xml/release/dblp-2023-06-01.xml.gz

[15] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. 2010. An Optimal Decomposition Algorithm for Tree Edit Distance. *ACM Trans. Algorithms* 6, 1, Article 2 (dec 2010), 19 pages. https://doi.org/10.1145/1644015.1644017

[16] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2012. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *2012 41st International Conference on Parallel Processing*. 350–359. https://doi.org/10.1109/ICPP.2012.19

[17] Antonio J. Dios, Rafael Asenjo, Angeles Navarro, Francisco Corbera, and Emilio L. Zapata. 2011. High-level template for the task-based parallel wavefront pattern. In *2011 18th International Conference on High Performance Computing*. 1–10. https://doi.org/10.1109/HiPC.2011.6152717

[18] Serge Dulucq and Hélène Touzet. 2003. Analysis of Tree Edit Distance Algorithms. In *Combinatorial Pattern Matching*, Ricardo Baeza-Yates, Edgar Chávez, and Maxime Crochemore (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–95.

[19] Serge Dulucq and Hélène Touzet. 2005. Decomposition algorithms for the tree edit distance problem. *Journal of Discrete Algorithms* 3, 2 (2005), 448–471. https://doi.org/10.1016/j.jda.2004.08.018 Combinatorial Pattern Matching (CPM) Special Issue.

[20] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. NestGPU: Nested Query Processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1008–1019. https://doi.org/10.1109/ICDE51399.2021.00092

[21] Zvi Galil and Kunsoo Park. 1994. Parallel Algorithms for Dynamic Programming Recurrences with More Than O(1) Dependency. *J. Parallel and Distrib. Comput.* 21, 2 (1994), 213–222. https://doi.org/10.1006/jpdc.1994.1053

[22] Henry Gilbert, Michael Sandborn, Douglas C. Schmidt, Jesse Spencer-Smith, and Jules White. 2023. Semantic Compression With Large Language Models. arXiv:2304.12512 [cs.AI]

[23] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. 2002. Approximate XML Joins. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/564691.564725

[24] Holger Heumann and Gabriel Wittum. 2009. The tree-edit-distance, a measure for quantifying neuronal morphology. *Neuroinformatics* 7 (2009), 179–190.

[25] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S Vetter, and Seyong Lee. 2018. Highly efficient compensation-based parallelism for wavefront loops on gpus. In *2018 IEEE International parallel and distributed processing symposium (IPDPS)*. IEEE, 276–285.

[26] Yuchi Huo, Shi Li, Yazhen Yuan, Xu Chen, Rui Wang, Wenting Zheng, Hai Lin, and Hujun Bao. 2022. ShaderTransformer: Predicting Shader Quality via One-Shot Embedding for Fast Simplification. In *ACM SIGGRAPH 2022 Conference Proceedings* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 44, 9 pages. https://doi.org/10.1145/3528233.3530722

[27] Di Jin, Zhijing Jin, Zhiting Hu, Olga Vechtomova, and Rada Mihalcea. 2022. Deep learning for text style transfer: A survey. *Computational Linguistics* 48, 1 (2022), 155–205.

[28] Nikolai Karpov and Qin Zhang. 2022. SyncSignature: A Simple, Efficient, Parallelizable Framework for Tree Similarity Joins. *Proc. VLDB Endow.* 16, 2 (oct 2022), 330–342. https://doi.org/10.14778/3565816.3565833

[29] Geewook Kim, Teakgyu Hong, Moonbin Yim, JeongYeon Nam, Jinyoung Park, Jinyeong Yim, Wonseok Hwang, Sangdoo Yun, Dongyoon Han, and Seunghyun Park. 2022. Ocr-free document understanding transformer. In *European Conference on Computer Vision*. Springer, 498–517.

[30] Philip N. Klein. 1998. Computing the Edit-Distance between Unrooted Ordered Trees. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, Berlin, Heidelberg, 91–102.

[31] Lambda Labs. 2024. Lambda Labs. https://lambdalabs.com/. Accessed: 2024-01-10.

[32] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (jun 2020), 38 pages. https://doi.org/10.1145/3383458

[33] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.* 14, 12 (jul 2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[34] Fei Li, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2014. A Survey on Tree Edit Distance Lower Bound Estimation Techniques for Similarity Join on XML Data. *SIGMOD Rec.* 42, 4 (feb 2014), 29–39. https://doi.org/10.1145/2590989.2590994

[35] Yuanzhe Li. 2020. Tiling Optimization For Nested Loops On Gpus. In *Wayne State University Dissertations*. 2362. https://doi.org/oa_dissertations/2362

[36] Yuanzhe Li and Loren Schwiebert. 2020. Memory-Optimized Wavefront Parallelism on GPUs. *International Journal of Parallel Programming* 48, 6 (2020), 1008–1031. https://doi.org/10.1007/s10766-020-00658-y

[37] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1238–1250. https://doi.org/10.1109/ICSE48619.2023.00110

[38] Malcolm Yoke Hean Low, Weiguo Liu, and Bertil Schmidt. 2023. A Parallel BSP Algorithm for Irregular Dynamic Programming. In *Advanced Parallel Processing Technologies: 7th International Symposium, APPT 2007 Guangzhou, China, November 22-23, 2007 Proceedings* (Guangzhou, China). Springer-Verlag, Berlin, Heidelberg, 151–160. https://doi.org/10.1007/978-3-540-76837-1_19

[39] Lixiang Luo, Jack Edwards, Hong Luo, and Frank Mueller. 2015. Optimization of A Fine-grained BILU by CUDA Inter-block Synchronization. https://doi.org/10.2514/6.2015-3055

[40] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2014. Parallelizing Dynamic Programming through Rank Convergence. *SIGPLAN Not.* 49, 8 (feb 2014), 219–232. https://doi.org/10.1145/2692916.2555264

[41] N. Manjikian and T.S. Abdelrahman. 2001. Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 12, 3 (2001), 259–271. https://doi.org/10.1109/71.914756

[42] Xiao Mao. 2021. Breaking the Cubic Barrier for (Unweighted) Tree Edit Distance. arXiv:2106.02026 [cs.DS]

[43] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).

[44] Benjamin Paaßen. 2018. Revisiting the tree edit distance and its backtracing: A tutorial. *CoRR* abs/1805.06869 (2018). arXiv:1805.06869 http://arxiv.org/abs/1805.06869

[45] Benjamin Paaßen, Claudio Gallicchio, Alessio Micheli, and Barbara Hammer. 2018. Tree Edit Distance Learning via Adaptive Symbol Embeddings. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 3976–3985. https://proceedings.mlr.press/v80/paassen18a.html

[46] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: A Robust Algorithm for the Tree Edit Distance. *Proc. VLDB Endow.* 5, 4 (dec 2011), 334–345. https://doi.org/10.14778/2095686.2095692

[47] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173. https://doi.org/10.1016/j.is.2015.08.004

[48] Mateusz Pawlik and Nikolaus Augsten. 2020. Minimal Edit-Based Diffs for Large Trees. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 1225–1234. https://doi.org/10.1145/3340531.3412026

[49] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. arXiv:2201.11227 [cs.LG]

[50] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (oct 2016), 731–747. https://doi.org/10.1145/3022671.2984041

[51] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1109/ICSE.2017.44

[52] Abhinav Sarje and Srinivas Aluru. 2010. A MapReduce style framework for computations on trees. In *2010 39th International Conference on Parallel Processing*. IEEE, 343–352.

[53] Stefan Schwarz, Mateusz Pawlik, and Nikolaus Augsten. 2017. A New Perspective on the Tree Edit Distance. In *Similarity Search and Applications*, Christian Beecks, Felix Borutta, Peer Kröger, and Thomas Seidl (Eds.). Springer International Publishing, Cham, 156–170.

[54] Masoud Seddighin and Saeed Seddighin. 2022. 3+ $\varepsilon$ Approximation of Tree Edit Distance in Truly Subquadratic Time. In *13th Innovations in Theoretical Computer Science Conference (ITCS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[55] Parijat Shukla and Arun K. Somani. 2015. Tree Matching Using Data Shaping. In *2015 IEEE International Congress on Big Data*. 166–173. https://doi.org/10.1109/BigDataCongress.2015.32

[56] José Maria Simões, Nuno Lourenço, and Penousal Machado. 2023. All You Need is Sex for Diversity. In *Genetic Programming*, Gisele Pappa, Mario Giacobini, and Zdenek Vasicek (Eds.). Springer Nature Switzerland, Cham, 276–291.

[57] Vivek Sourabh, Parth Pahariya, Isha Agarwal, Ankit Gautam, and C. Ravindranath Chowdary. 2017. Parallel Implementation of Dynamic Programming Problems Using Wavefront and Rank Convergence with Full Resource Utilization. In *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 151–155. https://doi.org/10.1109/PDCAT.2017.00033

[58] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. 2010. Lock-Free Parallel Dynamic Programming. *J. Parallel Distrib. Comput.* 70, 8 (aug 2010), 839–848. https://doi.org/10.1016/j.jpdc.2010.01.004

[59] Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (jul 1979), 422–433. https://doi.org/10.1145/322139.322143

[60] Guangming Tan, Shengzhong Feng, and Ninghui Sun. 2006. Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 41–41. https://doi.org/10.1109/SC.2006.41

[61] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. 2003. Phoenix: A Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 216–229. https://doi.org/10.1145/781498.781533

[62] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating Wavefront Parallelization for Sparse Matrix Computations. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 480–491. https://doi.org/10.1109/SC.2016.40

[63] Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Yulei Sui, Hongyu Zhang, Kazuma Hashimoto, Hai Jin, Guandong Xu, Caiming Xiong, and Philip S. Yu. 2022. NaturalCC: An Open-Source Toolkit for Code Intelligence. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 149–153. https://doi.org/10.1145/3510454.3516863

[64] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[65] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H. Saltz. 2012. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1543–1554. https://doi.org/10.14778/2350229.2350268

[66] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (1986), 279–293. https://doi.org/10.1007/BF01407876

[67] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470477

[68] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 229–238. https://doi.org/10.1145/2442516.2442539

[69] Weixiang Yan and Yuanchun Li. 2022. WhyGen: Explaining ML-Powered Code Generation by Referring to Training Examples. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 237–241. https://doi.org/10.1145/3510454.3516866

[70] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. 2005. Similarity Evaluation on Tree-Structured Data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 754–765. https://doi.org/10.1145/1066157.1066243

[71] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (aug 2013), 817–828. https://doi.org/10.14778/2536206.2536210

[72] Haoran Zhang and Li Li. 2023. HCPG: a highlighted contrastive learning framework for exemplar-guided paraphrase generation. *Neural Computing and Applications* 35, 23 (2023), 17267–17279. https://doi.org/10.1007/s00521-023-08609-7

[73] Kaizhong Zhang. 1996. Efficient parallel algorithms for tree editing problems. In *Combinatorial Pattern Matching*, Dan Hirschberg and Gene Myers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 361–372.

[74] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. https://doi.org/10.1137/0218082

[75] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1226–1237. https://doi.org/10.14778/2809974.2809984

[76] Simon Zhang, Mengbai Xiao, and Hao Wang. 2020. GPU-accelerated computation of Vietoris-Rips persistence barcodes. *arXiv preprint arXiv:2003.07989* (2020).