Efficient Accelerated Graph Edit Distance Computation on GPU

Adel Dabah[1] and Andreas Herten[1]

Jülich Supercomputing Center Forschungszentrum Jülich Jülich, Germany
a.dabah@fz-juelich.de a.herten@fz-juelich.de

Abstract. Graph representation is a powerful abstraction of real-world objects and relations. Computing the Graph Edit Distance (GED) between graphs is critical in domains such as bioinformatics, machine learning, and pattern recognition. GED measures the minimum number of edit operations required to transform one graph into another. However, the high computational complexity of optimal and near-optimal methods limits their applicability to large-scale graphs, making high-performance parallel GED computation essential. To address this, we propose FAST-GED, a fast and scalable open-source framework for GED computation on GPUs. FAST-GED overcomes existing limitations by combining high accuracy with fast execution through GPU-friendly algorithmic design and efficient mapping to GPU hardware, minimizing host-device communication. The implementation is optimized and tested across multiple GPU architectures. We validate FAST-GED on real and synthetic datasets with diverse graph sizes and densities. It achieves speedups of several orders of magnitude over the Python NetworkX library while reaching optimal solutions in most cases. Moreover, it outperforms state-of-the-art approximate methods in both accuracy and scalability. We show that FAST-GED enables broader adoption of GED-based solutions in real-world applications.

Keywords: Graph Edit Distance · GPU Computing · Parallel Algorithms · Graph Matching · High-Performance Computing

1 Introduction

Graph computations are essential for problems in social networks, cybersecurity, logistics, and machine learning. Large real-world graphs require high computational power and scalability, making High-Performance Computing (HPC) crucial for advancing graph-based applications. Introduced by Sanfeliu and Fu in 1983 [21], Graph Edit Distance (GED) finds the minimum cost set of edit operations: substitution, insertion, and deletion—needed to transform one graph into another [4]. The cost of each operation can be adapted per application, making GED a flexible and adaptive tool.

The GED problem is NP-hard, as the number of possible solutions grows exponentially with the number of vertices [8]. Existing methods are computationally intensive, especially for large graphs, and can be categorized into exact and

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

approximate algorithms. Exact approaches such as Depth-First Search (DFS) and A-Star guarantee optimality but are impractical for large datasets. Approximate methods, including K-Best [7, 8], VF2 [5], and Beam Search (BS) [13], improve efficiency at the cost of accuracy. Yet, scalability and accuracy remain major challenges, particularly for large graphs.

To address these limitations, we propose FAST-GED, a novel GPU-accelerated framework for GED computation that surpasses existing methods in both speed and accuracy. Leveraging the parallel processing capabilities of GPUs [15], FAST-GED overcomes the trade-off in K-Best methods, where scalability and accuracy are constrained by the parameter K. Our framework efficiently maintains both by exploiting GPU parallelism for large-scale GED computation.

FAST-GED operates on a search tree representing all possible edit paths transforming a source graph into a target graph. The tree is built via vertex-based branching, with each node decomposing into smaller subproblems. Similar to Breadth-First Search (BFS), our method explores levels of the search tree, retaining only the best K nodes at each level. After transferring graph data to the GPU, FAST-GED executes three main kernels: A branching kernel where each GPU block expands one node and computes its partial edit distances (PEDs). A ranking phase to select the best K child nodes without full sorting, using a two-step local and global ranking via atomic operations. An update phase that refreshes data structures with the best nodes, avoiding host-device communication and ensuring scalability. This process repeats until the final level, yielding the minimum-cost edit path.

We evaluated FAST-GED against state-of-the-art methods on real and synthetic datasets with varying graph sizes and densities. Compared to the Python NetworkX library [11], FAST-GED achieves optimal accuracy in over 90 % of cases with an order-of-magnitude lower execution time. On real-world datasets, it consistently outperforms Beam Search (BS) [13] and DFS-1 [1]. Furthermore, FAST-GED achieves a 300× speedup on an NVIDIA A100 GPU compared to a parallel CPU version on a 48-core AMD EPYC baseline, demonstrating the strong potential of GPU acceleration for GED computation.

The rest of this paper is organized as follows. Section 2 introduces the problem formulation and notations. Section 3 reviews related work. Section 4 presents the design and implementation of FAST-GED. Section 5 discusses experimental results. Section 6 showcase two important applications of FAST-GED, and section 7 concludes the paper.

2 Problem Formulation

In the following, we first define some basic concepts and then define the GED problem formally.

2.1 Graph

A graph is a structure used to model pairwise relations between objects [22]. It contains a set of vertices connected by a set of edges. In this paper, we consider

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

simple undirected labelled graphs, i.e., at most one edge between vertices and without loops. Formally, a labeled graph is denoted by $G = (V, E, \alpha, \beta)$, where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n vertices.
- $E = \{e_1, e_2, \ldots, e_m\}$ is a set of m edges ($E \subseteq V \times V$).
- α is a labeling function for the vertices, $\alpha : V \to L_V$.
- β is a labeling function for the edges, $\beta : E \to L_E$.

Here, L_V and L_E represent the sets of possible labels for vertices and edges, respectively, which can be integers, real-valued vectors, text, or symbolic labels.

2.2 Graph edit distance

Given two graphs $g_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $g_2 = (V_2, E_2, \alpha_2, \beta_2)$, the graph edit distance $d(g_1, g_2)$ measures the dissimilarity between them. It is defined as the minimum cost of the edit operations needed to transform g_1 into g_2:

$$d(g_1, g_2) = \min_{\{o_1, o_2, \ldots, o_k\} \in \gamma(g_1, g_2)} \sum_{i=1}^{k} c(o_i)$$

where $\gamma(g_1, g_2)$ denotes the set of all possible ways transforming g_1 into g_2 (edit paths) , and $c(o_i)$ is the cost of the edit operation o_i [18].

2.3 GED Operations

The GED problem involves transforming one graph g_1 into another graph g_2 through a series of edit operations. In this work, we consider a vertex-centric approach in which edit operations are performed on vertices, and edges are implied. Let us consider a vertex $v_i \in V_1$ from g_1 and a vertex $u_j \in V_2$ from g_2, we denote the three edit operations:

1. **Substitution**: $v_i \to u_j$, v_i is substituted by u_j.
2. **Deletion**: $v_i \to \epsilon$, v_i is deleted from g_1.
3. **Insertion**: $\epsilon \to u_j$, u_j is inserted into g_1.

Implied edges operations In the vertex-centric approach, edge operations are implied [18]. Therefore, implied edge operations are not applied as independent edit steps on intermediate graphs. Instead, whenever a new vertex operation is applied to the current edit path λ, we check its incident edges with all previously edited vertices in λ and compute the corresponding edge costs (substitution, insertion, or deletion). Let us consider two vertices v, v' from graph g_1 and two other vertices u, u' from graph g_2 and $\lambda = \{..., v \to u\}$. If we perform the following edit operation $\{v' \to u'\}$, three cases of implied edges exist:

1. If there is an edge $e_1(v, v')$ in g_1 and there is also an edge $e_2 = (u, u')$ in g_2, then e_1 is substituted by e_2, denoted $e_1 \to e_2$.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

2. If there is an edge $e_1(v, v')$ in g_1 and there is no edge between u and u' in g_2, then e_1 is deleted from g_1, denoted $e_1 \rightarrow \epsilon$.
3. If there is no edge between v and v' in g_1 and there is an edge $e_2(u, u')$ between u and u' in g_2, then e_2 is inserted into g_1, denoted $\epsilon \rightarrow e_2$.

If a vertex is deleted from (resp. inserted to) g_1 all its incident edges are automatically deleted (resp inserted).

2.4 Cost Function

The cost function assigns a cost to each edit operation incorporating domain-specific information about the similarity of objects. The substitution cost β' is zero if the two vertices or edges have the same label. $c(u \rightarrow v) = c(v \rightarrow u) = \beta'$ / $c(u \rightarrow \epsilon) = \theta$ / $c(\epsilon \rightarrow v) = \theta$

3 Related Work

Graph-based representations address challenges such as image translation, rotation, and scaling [10], and GED is a widely used measure for exact and inexact graph matching. Since its introduction by Sanfeliu and Fu [21], GED has been solved using A-Star [4] and improved heuristics such as the bipartite heuristic [18–20]. Depth-First approaches [1] reduce memory usage, and parallel B&B implementations [6] exploit multi-threading and load balancing to accelerate GED computation. Exact methods remain impractical for large graphs due to the NP-hard nature of GED, motivating approximate strategies. These include parallel fixed-time Branch & Bound [9], Beam Search [13], and our K-Best tree-based approach [7], these methods lack accuracy and are not scalable for large graphs. FAST-GED addresses these limitations, by efficient algorithmic design exploiting efficiently GPU computing power.

4 FAST-GED: GPU-Accelerated Graph Edit Distance

The FAST-GED approach mimics the behavior of a Breadth-First Search (BFS) algorithm in traversing a search tree. The approach explores the tree level by level until reaching leaf nodes, performing both branching and evaluation for all nodes at each level before advancing to the next one. However, unlike traditional BFS, the FAST-GED approach selectively considers only the best K nodes, based on evaluation criteria, for further exploration at the next level. Intuitively, the accuracy of this approach improves as the parameter K increases. This selective exploration process involves two distinct phases: a branching phase, where all nodes at a given level undergo branching and evaluation, and a selection phase, where the best K successor nodes are chosen for advancement to the next level based on their evaluation. These phases are repeated iteratively until the final level containing potential solutions is reached. Hence, the best solution is returned as an approximate solution for the GED problem. The best nodes selected at each level increase the likelihood of capturing the optimal path within the search space. Algorithm 1 illustrates the different phases of this approach.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

4.1 GPU Architecture

A Graphics Processing Unit (GPU) is an accelerator initially designed for gaming and optimized for parallel computation. GPU parallelism follows the Single Instruction Multiple Thread (SIMT) paradigm, where multiple threads execute concurrently. Threads are organized into blocks, and blocks form grids. Threads within a block can share data via shared memory and synchronize execution using barriers. GPUs also feature multiple memory hierarchies, including global, constant, texture, register and cache memory, each with specific access patterns. Making efficient use of these memory types is crucial for maximising GPU performance.

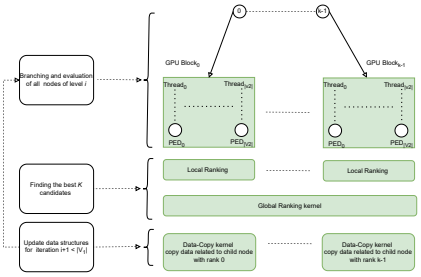


Fig. 1: GPU-based parallelization of FAST-GED: Mapping of search tree expansion and ranking phases onto GPU blocks and threads.

Algorithm 1: FAST-GED algorithm

Data: Attributed graphs $g_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $g_2 = (V_2, E_2, \alpha_2, \beta_2)$ where $V_1 = \{u_1, ..., u_{|V_1|}\}$ and $V_2 = \{v_1, ..., v_{|V_2|}\}$

Result: Approximate graph edit distance and corresponding edit path

```
1  List ← {initial problem};
2  best_path ← null;
3  foreach (v ∈ V1) do
4      foreach (node ∈ List) do
5          Generate nd_j, where
             j ∈ [1, ..., |R_V2| + 1];
6          foreach successor(nd_j) do
7              calculate PED(nd_j);
8              list_tmp ← add_node(nd_j);
9      List ← {best K nodes in list_tmp}
10 best_path ← λ (node with best PED in
     List);
11 return best_path;
```

4.2 Phase 1: Branching and Evaluation

Branching This phase generates successors of all nodes of a given level by decomposing each problem into several smaller sub-problems. The branching is based on the vertex-centric approach, where a vertex from the first graph g_1 is mapped to the remaining vertices from g_2 using substitution, deletion, and insertion operations. A search tree node nd is characterized by:

- A *path* $\lambda = \{o_1, o_2, ..., o_k\}$ where o_i represents past edit operations.
- Remaining vertices from g_1 and g_2 denoted by R_{V1} and R_{V2} respectively.

In this way, the root node is defined as $|R_{V1}| = |V_1|$, $|R_{V2}| = |V_2|$, $\lambda = \emptyset$. The branching on a search tree node nd generates $|R_{V2}|$ new successors by performing the substitution of a vertex $v \in R_{V1}$ with all vertices in R_{V2}. In other words, each successor node nd_j (where $j \in [1..|R_{V2}|]$) represents the mapping of a vertex $v \in$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

R_{V1} to a vertex $u \in R_{V2}$ and is defined as follows: nd_j $\{R_{V1}(nd_j) = R_{V1}(nd) \setminus \{v\}$; $R_{V2}(nd_j) = R_{V2}(nd) \setminus \{u\}$; $\lambda(nd_j) = \lambda(nd) \cup \{v \rightarrow u\}\}$. In addition, we add a new successor that represents the deletion of the vertex $v \in R_{V1}$. $R_{V1}(nd_j) = R_{V1}(nd) \setminus \{v\}$; $R_{V2}(nd_j) = R_{V2}(nd)$; $\lambda(nd_j) = \lambda(nd) \cup \{v \rightarrow \epsilon\}$.;

Evaluation The evaluation process calculates the Partial Edit Distance (PED) for each successor node. i.e, Cost of all edit operations in $\lambda(nd_j)$: $PED(nd_j) = \sum_{i=1}^{|y|} c(o_i)$.

4.3 Phase 2: Selection

This step aims to select nodes more likely to contain promising paths. After evaluating all resulting nodes from the first phase, we select the K best nodes based on their evaluations. Practically, this step involves ranking the nodes based on their evaluation and then removing all nodes ranked greater than K. This ranking process can be computationally intensive, particularly when dealing with a large number of nodes, which is challenging for real-time applications. To enhance the accuracy of this approach, a large value of K is essential, as the differences in node evaluations are minimal, especially in the early stages of the search tree. However, this will linearly increase the complexity. To overcome this issue, we leverage the computational power of GPUs.

4.4 GPU Implementation

This section describes the GPU implementation of the FAST-GED algorithm.

First Phase of FAST-GED To parallelize the first phase of the FAST-GED algorithm, which involves the generation and evaluation of successors on the GPU, we opted for the following mapping. As shown in algorithm 2, each GPU block will handle the branching and evaluation of a node, such that each thread on the GPU will be responsible for generating and evaluating one successor. This involves the mapping of a vertex v from the first graph g_1 to the remaining vertices in g_2 using the substitution operation. The total number of block threads is equal to the number of vertices in g_2; i.e., thread i always performs the substitution operation using u_i. In addition, one thread will be responsible for the deletion operation. Note that vertex insertions are handled at the end of the search process: once all vertices of g_1 have been mapped or deleted, any remaining vertices in g_2 are inserted, and the corresponding costs are added.

The second part of this phase involves evaluating all successor nodes, which is done in two steps. The first step is common to all threads in a block, and the second step is specific to each thread. In the common first step, we calculate the implied edges following the mapping of vertex v in g_1. This requires going through all previous edit operations o_i in λ. For each edit operation o_i, we check if there is an edge between v and the source or destination vertices of o_i. The results are stored in two vectors, $VFrom$ and VTo, in shared memory.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

Algorithm 2: First Phase of FAST-GED on GPU

Input: Two graphs g_1 and g_2
Output: Successor nodes evaluated
Initialization: Nb GPU blocks $\leftarrow k$;
Nb threads per block $\leftarrow |V_2|$ Nb vertices in g_2;
Parallel Execution:
foreach block b **do**
  get vertex v in R_{V1} from $node_b$;
  foreach thread t in block b **do**
    if $u_t \in R_{V2}$ **then**
      substitute vertex v with u_t in g_2 ;
      $Imp\_cost = Implied\_Edge\_Cost()$;
      $PED = E(node_b) + c(v \leftarrow u_t) + Imp\_cost$;
    end
  end
end

Next, in the thread-specific second step, each thread t examines all edit operations in λ. For each o_i, we check if there is an edge between u_t and the source or destination vertices of o_i. For each edit operation, we consider three cases: If an edge exists in g_1 ($VFrom[i] \neq$ Null or $VTo[i] \neq$ Null) but not in g_2, we add an edge deletion cost. If an edge exists in g_2 but not in g_1, we add an edge insertion cost. If the edge exists in both graphs but with different weights, we add an edge substitution cost.

Finally, the partial edit distance is computed for each successor node evaluation, which is the sum of the parent partial edit distance, the cost of vertex mapping, and the implied edges cost.

Second Phase of FAST-GED The second phase of the FAST-GED algorithm involves selecting the best K candidates for the next iteration. While a straightforward approach would be to perform sorting and select the top K nodes, sorting algorithms can be computationally expensive and introduce significant overhead, particularly for scalability purposes. However, we do not require a sorted list; rather, we only need the top K candidates in a non-sorted order. We do not perform a full sort of all candidates. Instead, we directly extract the top-K nodes using block-local selection and global thresholding. The final set of K candidates is therefore unordered. Initially, we observed a lack of efficient algorithms capable of performing such operations on the GPU. Therefore, we propose the following approach to address this issue. After evaluation, we determine the local ranking of threads within each block using shared memory to reduce latency and facilitate data sharing among all threads in a block. The top L threads from each block then update a global list of the best L Partial Edit Distance (PED) values using atomic operations. Limiting the operation to the best threads helps mitigate the overhead associated with atomic operations.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

Subsequently, a second kernel is employed to assign global rankings to threads. Threads with evaluations equal to the highest value are assigned the top ranks. Subsequent threads with evaluations equal to the second-highest value are ranked thereafter, continuing until the top K candidates are identified.

Finally, we update the data structures with the values for the next iteration, thereby avoiding any host-device communication. During experimentation, we set $L = 5$; however, we have the flexibility to adjust this value without a significant time increase.

4.5 Complexity Analysis

In the following, we analyze the time and space complexity of FAST-GED. Let $n = \max(|V_1|, |V_2|)$ and K be the number of nodes retained per level. Note that as $K \to \infty$, FAST-GED becomes equivalent to exhaustive breadth-first search and therefore converges to the optimal edit distance.

Time complexity: At level i ($0 \leq i < n$), each of the K retained nodes branches to $|RV_2| + 1$,*i.e.*, $n + 1$ successors in worst case (substitutions with all remaining vertices in RV_2, plus one deletion). Computing the partial edit distance for a successor requires iterating over the current edit path λ (of size $\leq i \leq n$) to account for implied edges, yielding $\mathcal{O}(n)$ per successor. Sequentially this gives $\mathcal{O}(K \cdot n \cdot n) = \mathcal{O}(Kn^2)$ per level and $\mathcal{O}(Kn^3)$ overall across n levels.

When performing the parallelization on the GPU, we map one node to a GPU block where each thread handles one successor. This collapses the successor loop, and each block performs an $\mathcal{O}(n)$ shared pass over λ, so the per-node cost is $\mathcal{O}(n)$. With sufficient parallelism to process the K nodes concurrently, the per-level time is $\mathcal{O}(n)$, leading to an overall complexity of $\mathcal{O}(n^2)$.

The top-K selection has a linear complexity due to local rankings and global atomic operations rather than full sorting. It can become a bottleneck when K is extremely large, in this case, approximate top-k selection may become an option.

Space complexity: The algorithm space complexity is $\mathcal{O}(K \cdot n)$. Indeed, each of the K nodes at a given level stores an edit path λ and two sets of remaining vertices, each of size $\mathcal{O}(n)$. Therefore, the total memory grows linearly with both K and n. In practice, this trade-off between accuracy (through K) and memory cost is handled efficiently using GPU memory hierarchies, data reuse, and parallel reductions.

5 Performance Evaluation

This section evaluates the accuracy and scalability of FAST-GED using standard datasets and state-of-the-art methods. The source code (JAVA/CUDA) and datasets are available in the FAST-GED repository[1]. We define g_1 as the source and g_2 as the target throughout all experiments for consistency. Vertex substitution, insertion, and deletion costs are set to 2, 4, and 4, respectively,

[1] `https://gitlab.jsc.fz-juelich.de/dabah2/fast-ged/-/tree/main`

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

while edge substitution, insertion, and deletion costs are 1, 2, and 2. These settings favor substitution over insertions and deletions to avoid trivial edit paths, but other cost functions are also supported. All experiments are performed on a node with four NVIDIA A100 GPUs and dual AMD EPYC 7702 CPUs (i.e. 48 cores). Unless stated otherwise, the parameter K is fixed to 700,000.

We first validate FAST-GED by comparing its deviation from the optimal results of the NetworkX library [11], a Python package for complex graph analysis that uses a DFS strategy with the bipartite heuristic [18] to estimate future path costs. Table 1 summarizes our results for small random graphs (ten vertices) across different graph densities, reporting average edit distance, deviation from optimal NetworkX results, speedup, and the number of times FAST-GED reached optimal matches. Since NetworkX library requires several minutes per GED, a 100 graph combinations for each density represents a good balance between computational effort and analysis.

Method/Density	**0.1**	**0.3**	**0.5**	**0.7**	**0.9**
NetworkX	17.95	19.21	19.97	20.83	22.51
FAST-GED	18.05	19.35	20.12	20.99	22.75
Deviation (%)	0.55	0.71	0.65	0.71	1.00
Speedup	26×	38×	48×	50×	55×
Optimal (%)	94/100	92/100	95/100	92/100	90/100

Table 1: Comparison of NetworkX library optimal GED results and FAST-GED approach using random graphs at different densities.

FAST-GED achieves less than 1% deviation from NetworkX's optimal results, while demonstrating substantial speedup improvements, ranging from 26× to 55×. The speedup increase with graph density arises from the higher execution time of the NetworkX algorithm, whose bipartite heuristic has a complexity of $\mathcal{O}(n^3)$. In sparse graphs, the simpler bipartite structure allows near $\mathcal{O}(n^2)$ performance and effective pruning, whereas in dense graphs, the structure becomes more complex, reducing pruning efficiency and increasing computational cost. Unlike the NetworkX, FAST-GED's computational complexity remains agnostic to graph density, ensuring consistent performance across varying graph structures. Over 500 GED computations using random graphs, FAST-GED consistently achieves the optimal edit distance in more than 90 % of cases, showcasing its near optimal performance.

Table 2 presents the mean edit distance of FAST-GED and some of the best approximate approaches using medium-sized real-world datasets. The CMU dataset [12], created by Carnegie Mellon University's Robotics Institute. GREC and MUTA are subsets of the IAM graph database repository proposed in [17]. The size column denotes graph size, and NB refers to the number of tested combinations. We consider two baselines: the Beam Search (BS) method [13],

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

which limits its priority queue size to ten elements to balance accuracy and execution time; and DFS_1 [2], one of the few approaches that scales. For each approach, we report the mean edit distance for all 55 graph combinations (lower is better). Optimal B&B approaches cannot handle medium size graphs in a reasonable amount of time [8].

Dataset	Size	NB	FAST-GED	BS_{10}	DFS_1	BS Time (s)	DFS time (s)	FAST-GED time (s)
GREC	20	55	32.0	37	39.8	1.7	1.0	1.0
	Mix	55	25.4	36.9	28.3	1.4	1.0	1.0
CMU	30	55	95.9	132.1	171.5	239.0	1.0	1.0
MUTA	20	55	17.9	23.6	31.0	1.1	1.0	1.0
	30	55	25.4	36.8	41.8	6.4	1.0	1.0
	40	55	43.1	49.9	62.7	27.4	1.0	1.0
	50	55	50.5	68.8	77.7	49.6	1.0	1.0
	60	55	65.4	79.5	86.6	250.0	1.0	1.0
	70	55	73.6		113.5	600	1.0	1.0
	Mix	55	86.6	140.8	100.9	297.0	1.0	1.0

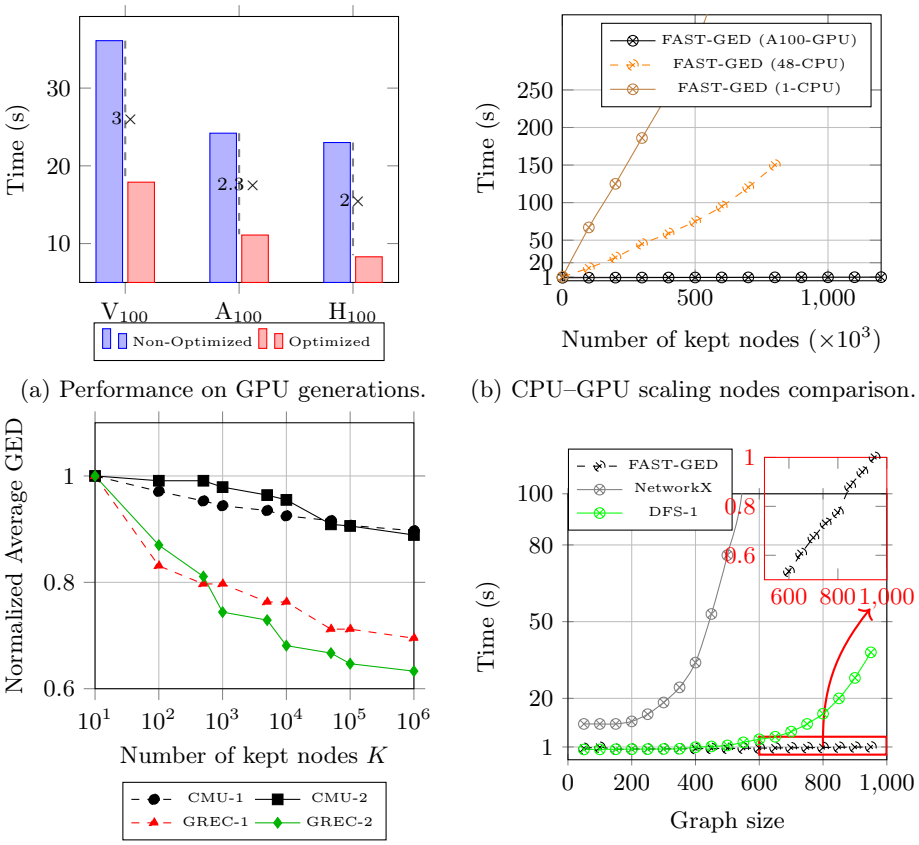
Table 2: Comparison of FAST-GED and state-of-the-art approximate methods on medium-sized real-world datasets.

Table 2 results confirm FAST-GED superiority, showing a lower mean edit distance compared to BS and DFS_1 across all datasets. Regarding time to solution, DFS_1 achieves the fastest execution time, followed by FAST-GED, with both approaches having an execution time of less than 1 second. Compared to BS_{10}, FAST-GED is up to 400 times faster for large datasets, while maintaining higher accuracy. Indeed, retaining more candidate nodes per search-tree level increases the likelihood of FAST-GED capturing the optimal path. GPU parallelism enables this broader exploration at low cost, whereas CPU methods would require hours for similar results.

Figure 2a shows the performance of optimized and non-optimized FAST-GED across different GPU architectures. The optimized version is 2× faster, with both versions using shared memory. The main optimization targets the final step of FAST-GED, which consists of preparing the data for the next iteration. Each thread previously copies parent-node data through three sequential loops, causing GPU-thread divergence and scattered memory accesses causing performance degradation. As a result, this step accounted for 40 % of the total runtime. The optimized version introduces a new *copy_kernel* in a block-wise fashion, where each thread has only three coalesced global memory accesses, reducing this step to only 5 % of the total runtime.

The A100 GPU achieves 40 % better performance than V100, due to the improved architecture, memory bandwidth and core count. Similarly, the H100 (PCIe 80 GB) yields a 55 % performance gain compared to V100 and 28 % over

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2



(a) Performance on GPU generations.

(b) CPU–GPU scaling nodes comparison.

(c) Effect of K on FAST-GED accuracy.

(d) Scalability with increasing graph size.

Fig. 2: Comprehensive performance evaluation of FAST-GED: (a) performance across GPU generations, (b) CPU–GPU scaling, (c) effect of node parameter K on accuracy, and (d) scalability with graph size.

A100. Indeed, the H100 has further architectural improvements, and FAST-GED did not fully benefit for it due to the synchronization and atomic operations needed to filter the top k candidates at each level of the tree, suggesting future optimizations should target reducing this overhead.

Figure 2b illustrates the runtime of FAST-GED serial, multi-core CPU, and GPU-based (A100) versions as the K increases using two small graphs with 20 vertices. The serial version runtime increases significantly due to the number of successors that need to be evaluated and sorting overhead. The multi-core CPU version generates and evaluates successors in parallel on 48 AMD CPU cores, resulting in a 4.5× relative speedup. This modest speedup is limited due to the final K-best selection step, which relies on a priority queue to merge sorted candidate lists from all threads. This introduces significant merging and

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

synchronization overhead, particularly for large values of K, where the overhead becomes dominant. On the other side, mapping and optimizing FAST-GED fully onto the GPU and avoiding any host-device data transfer allows retaining million of nodes while keeping runtime under one second with up to 300× faster runtime.

Figure 2c illustrates the advantage of increasing the number of nodes retained at each level of the FAST-GED search tree. We use the 55 graph combinations for both the CMU and GREC (20) datasets in Table 2 and report the average edit distance for all combinations. In addition, two cost-settings have been used for both the CMU and GREC datasets. **Setting 1:** $\{c_{\text{vsub}} = 2, c_{\text{vdel}} = c_{\text{vins}} = 4, c_{\text{esub}} = 1, c_{\text{edel}} = c_{\text{eins}} = 2\}$, where substitutions are cheaper, favoring mapping-based edits. **Setting 2:** $\{c_{\text{vsub}} = 4, c_{\text{vdel}} = c_{\text{vins}} = 12, c_{\text{esub}} = 1, c_{\text{edel}} = c_{\text{eins}} = 10\}$, where high insertion and deletion costs discourage structural changes. Average edit distances are normalized by their values at $K = 10$ for comparability. Across both datasets and cost settings, the same trend can be observed: increasing K reduces normalized edit distance, demonstrating improved accuracy. Two phases are distinguish: a rapid improvement for $K = 10$–1000, followed by slow decrease ($K > 1000$) as FAST-GED converges to the optimal edit distance as $K \to \infty$. Indeed, K serves as a tunable parameter that controls the trade-off between accuracy and efficiency, adjustable to the needs of each application.

Figure 2d illustrates the scalability of FAST-GED (K=5000) as the graph size increases. Used graphs were randomly generated with an average density of 0.4 using NetworkX. FAST-GED and DFS_1 exhibit similar scalability performance for graphs up to 600 vertices. Beyond this point, FAST-GED maintains a nearly linear runtime increase exploiting lightweight GPU threads, following the $\mathcal{O}(n^2)$ complexity analysis predicted in Section 4.5. Whereas DFS_1 shows a significant slowdown due to (i) the growing number of successors generated and evaluated sequentially and (ii) higher cost of traversing past edits in λ to compute implied edge costs.

Compared to the NetworkX approximate results, FAST-GED offers a faster and more scalable solution for GED computation, enabling its use in large-scale applications.

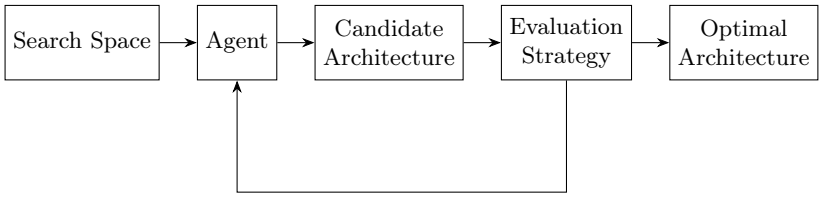
6 GED Applications

We demonstrate the effectiveness of FAST-GED through two relevant applications: Graph Classification and Neural Architecture Search (NAS).

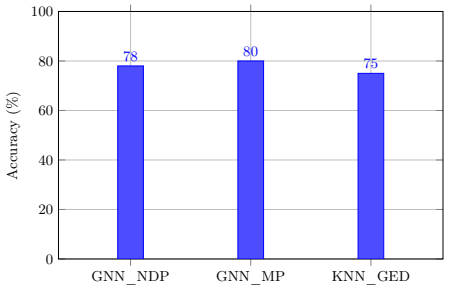
6.1 Graph Classification

Graph classification is a core machine learning task where models often struggle to capture structural similarity. Here, we combine FAST-GED with the K-Nearest Neighbor algorithm (**KNN_GED**) to enable a simple yet effective classification method based on structural distances. We use the Mutagenicity dataset [17], which classifies chemical compounds as mutagenic or non-

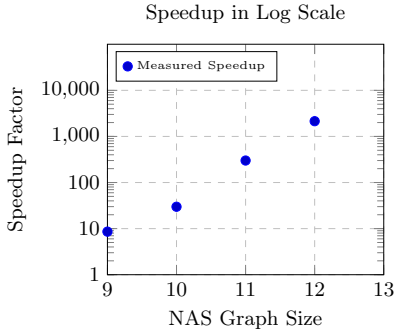
ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2



(a) Overview of Neural Architecture Search.



(b) Classification accuracy on the Mutagenicity dataset.



(c) FAST-GED speedup in architecture generation vs. NetworkX.

Fig. 3: Applications of FAST-GED. (a) Neural Architecture Search overview. (b) Graph classification accuracy using KNN_GED. (c) Speedup in NAS graph generation compared to NetworkX.

mutagenic. Graphs are split into 70 % training and 30 % testing sets. Each test graph is assigned to the class of its nearest training graph in terms of GED.

While computing thousands of pairwise GEDs using traditional approaches is prohibitive, taking weeks on large datasets, our GPU-accelerated framework reduces this to minutes, making KNN+GED practical and scalable. As shown in Figure 3b, KNN_GED achieves accuracy comparable to the best advanced Graph Neural Networks (GNN_MP [14], GNN_NDP [3]) while remaining interpretable and easy to implement. Using uniform GED costs (c_{ins}=c_{del}=2, c_{sub}=1) and the nearest neighbor (k=1), the method attains 75 % accuracy; tuning cost parameters can further improve results.

6.2 Neural Architecture Search (NAS)

Neural Architecture Search (NAS) is another machine learning field where GED can play an important role. As depicted in Figure 3a, the NAS goal is to explore the search space to automatically find the optimal architecture for a given problem that can outperform the manually designed architectures [23]. Neural architectures are represented as directed graphs, and exploring the search space efficiently involves applying operations on the graphs, like crossover and mutation in evolutionary search. GED can be used in the evaluation phase to avoid

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

evaluating similar tested architectures. This induces a huge gain since we avoid fitting a usually large amount of input data to the tested neural architecture, saving a lot of computing resources.

The second interesting application is in generating new candidate architectures by an efficient crossover operation based on GED [16]. Indeed, given two graphs, the edit path that transforms one graph into another can be used to generate new architectures by applying the transformation path to a certain number of edit operations, resulting in a new graph that combines the best parts of both parents. In other words, a new architecture C is generated from A and B by computing the GED edit path between them and applying half of its edit operations, producing a mixed graph of both. However, the final accuracy of the obtained model relies on both a near-optimal and fast GED computation, which was possible thanks to our open-source framework.

Figure 3c shows the speedup obtained by FAST-GED compared to NetworkX when generating ten new architectures with varying graph sizes. Our framework achieves up to $10^3\times$ acceleration with less than 10% deviation, well within the acceptable 30 % margin [16]. This drastic reduction in computation time—from hours to seconds—makes GED-based NAS strategies feasible at scale, enabling broader and more efficient exploration of neural architectures.

7 Conclusion

In this work, we presented FAST-GED, a GPU-based framework for efficient graph edit distance (GED) computation. By combining a tenable K-Best edit path search with an optimized GPU mapping that avoids host-device data transfer, FAST-GED achieves both high performance and scalability. Experimental results on real-world and synthetic datasets demonstrate significant speedups of 26–55$\times$ over NetworkX for small graphs, while consistently improving accuracy over state-of-the-art methods.

Future work will focus on extending FAST-GED to handle extremely large graphs with hundreds of thousands of vertices.

References

1. Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., Martineau, P.: An exact graph edit distance algorithm for solving pattern recognition problems. In: Pattern Recognition Applications and Methods Conference (2015)
2. Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., Martineau, P.: A distributed algorithm for graph edit distance. In: International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA). pp. 76–81 (2016)
3. Bianchi, F.M., Grattarola, D., Livi, L., Alippi, C.: Hierarchical representation learning in graph neural networks with node decimation pooling. IEEE Transactions on Neural Networks and Learning Systems **33**(5), 2195–2207 (2020)
4. Bunke, H., Allermann, G.: Inexact graph matching for structural pattern recognition. Pattern Recognition Letters **1**(4), 245–253 (1983)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2

5. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence **26**(10), 1367–1372 (2004)
6. Dabah, A., Bendjoudi, A., AitZai, A., Taboudjemat, N.N.: Efficient parallel tabu search for the blocking job shop scheduling problem. Soft Computing **23**(24), 13283–13295 (2019)
7. Dabah, A., Chegrane, I., Yahiaoui, S.: Efficient approximate approach for graph edit distance problem. Pattern Recognition Letters **134**, 46–57 (2021)
8. Dabah, A., Chegrane, I., Yahiaoui, S., Bendjoudi, A., Nouali-Taboudjemat, N.: Efficient parallel branch-and-bound approaches for exact graph edit distance problem. Parallel Computing **114**, 102984 (2022)
9. Dabah, A., Ltaief, H., Rezki, Z., Arfaoui, M.A., Alouini, M.S., Keyes, D.: Performance/complexity trade-offs of the sphere decoder algorithm for massive mimo systems. arXiv preprint arXiv:2002.09561 (2020)
10. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Analysis and Applications **13**(1), 113–129 (2010)
11. Hagberg, A.A., Schult, D.A., Swart, P.J.: Networkx. `https://networkx.org/` (2008), accessed: 2024-05-20
12. Kanade, T., Cohn, J.F., Tian, Y.: Comprehensive database for facial expression analysis. `http://www.cs.cmu.edu/~face/` (2000), cMU Robotics Institute
13. Neuhaus, M., Riesen, K., Bunke, H.: Fast suboptimal algorithms for the computation of graph edit distance. In: Joint IAPR International Workshops. pp. 163–172 (2006)
14. Nouranizadeh, A., Matinkia, M., Rahmati, M., Safabakhsh, R.: Maximum entropy weighted independent set pooling for graph neural networks. arXiv preprint arXiv:2107.01410 (2021)
15. NVIDIA Corporation: Nvidia cuda c programming guide. Tech. rep., NVIDIA Corporation (2023), version 12.3
16. Qiu, X., Miikkulainen, R.: Shortest edit path crossover: A theory-driven solution to the permutation problem in evolutionary neural architecture search. In: International Conference on Machine Learning. pp. 28422–28447 (2023)
17. Riesen, K., Bunke, H.: Iam graph database repository for graph based pattern recognition and machine learning. Structural, Syntactic, and Statistical Pattern Recognition pp. 287–297 (2008)
18. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image and Vision Computing **27**(7), 950–959 (2009)
19. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: MLG Workshop (2007)
20. Riesen, K., Fischer, A., Bunke, H.: Computing Upper and Lower Bounds of Graph Edit Distance in Cubic Time, pp. 129–140. Springer (2014)
21. Sanfeliu, A., Fu, K.S.: A distance measure between attributed relational graphs for pattern recognition. IEEE Transactions on Systems, Man, and Cybernetics **3**, 353–362 (1983)
22. West, D.B.: Introduction to Graph Theory. Prentice Hall (2001)
23. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29921-5_2