G-PathGen: An Efficient GPU-Parallel k-Critical Path Generation Algorithm

Che Chang
UW-Madison
Madison, USA
cchang289@wisc.edu

Yi-Hua Chung
UW-Madison
Madison, USA
yihua.chung@wisc.edu

Cheng-Hsiang Chiu
UW-Madison
Madison, USA
chenghsiang.chiu@wisc.edu

Wan-Luan Lee
UW-Madison
Madison, USA
wlee329@wisc.edu

Boyang Zhang
UW-Madison
Madison, USA
bzhang523@wisc.edu

Ulf Schlichtmann
Technical University of Munich
Munich, Germany
ulf.schlichtmann@tum.de

Ing-Chao Lin
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
iclin@nycu.edu.tw

Xiangyao Yu
UW-Madison
Madison, USA
yxy@cs.wisc.edu

Tsung-Wei Huang
UW-Madison
Madison, USA
tsung-wei.huang@wisc.edu

Abstract

Critical path generation (CPG) plays a key role in many circuit timing analysis (CTA) applications. As the design complexity continues to increase, CPG runtime has become a major bottleneck in many timing-driven applications. To mitigate this runtime challenge, several CPU-based algorithms have been introduced by both the CTA and parallel computing communities, but they remain slow for large CPG problems. While GPU-accelerated solutions exist, they are often inexact and incur significant overhead from iterative CPU–GPU data transfers, limiting their practical use in CTA applications. To overcome this challenge, we propose G-PathGen, an exact GPU-parallel CPG algorithm targeting CTA applications. G-PathGen introduces efficient kernel algorithms for generating critical paths in parallel and dynamically adjusts the generated path count to maximize GPU utilization while minimizing redundant work. Compared to a state-of-the-art GPU solution, G-PathGen is 1.6×–243.8× faster when generating one million critical paths on industrial circuit graphs.

ACM Reference Format:
Che Chang, Yi-Hua Chung, Cheng-Hsiang Chiu, Wan-Luan Lee, Boyang Zhang, Ulf Schlichtmann, Ing-Chao Lin, Xiangyao Yu, and Tsung-Wei Huang. 2026. G-PathGen: An Efficient GPU-Parallel k-Critical Path Generation Algorithm. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3797905.3800511

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '26, Belfast, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800511

1 Introduction

Critical path generation (CPG) is a crucial step for circuit timing analysis (CTA) applications to assess the timing criticality of a circuit design [5]. As the design complexity continues to increase, the runtime of CPG has quickly become a bottleneck in many CTA applications, such as timing-driven optimization and sign-off [37]. To mitigate this runtime challenge, the CTA community has proposed several single-threaded CPG algorithms that efficiently generate top-k critical paths from a circuit graph, such as branch-and-bound in iTimerC [66], pin coloring in iitRACE [90], and implicit path ranking in OpenTimer [42]. To further enhance the performance, researchers have proposed several multithreaded CPG algorithms, such as parallel tree contraction [91], hierarchical parallelism in PeeK [24], and multi-level queue scheduling in PathGen [8]. Despite the promising performance, these CPG algorithms are largely limited to CPU parallelism and remain slow for large CTA problems. For example, a CPG query of 1M paths on a circuit graph of 4M gates can take more than three seconds [42], while industrial CTA applications often issue thousands of CPG queries in a timing-driven optimization loop [89], which easily accumulates to hours.

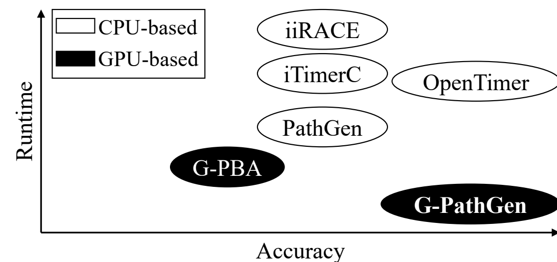

Figure 1: Runtime-accuracy tradeoff of state-of-the-art critical path generation algorithms in the circuit timing analysis community.

To further mitigate this runtime challenge, G-PBA [29] introduced a GPU-accelerated CPG algorithm that iteratively explores path candidates from a levelized graph structure. To avoid generating too many paths that may exhaust GPU memory, G-PBA evaluates path criticality and performs iterative path pruning on CPU. While G-PBA substantially outperforms CPU-based algorithms, it suffers from three major drawbacks that prevent it from practical adoption by CTA applications: (1) It does not yield exact k-critical paths, as early pruning may discard paths that later prove more critical. (2) It incurs significant overhead due to iterative data movement between CPU and GPU required by the pruning process. (3) Its kernel algorithm is a straightforward extension of the single-threaded CPG algorithm, which leaves substantial room for performance improvement.

However, designing an exact CPG algorithm on GPU is very challenging for three reasons: (1) We need to design a GPU-efficient data structure to handle parallel path generation. (2) GPU-parallel path generation can produce a massive number of paths beyond the GPU memory capability. To avoid this problem, we need a stepping strategy that can properly control the generated path count per step. (3) While controlling the generated path count per step helps alleviate GPU memory pressure, it is important to strike a balance between parallelism and work efficiency; too few paths underutilize the GPU, whereas too many result in wasted computation on non-k-critical paths.

To overcome these challenges, we propose *G-PathGen*, an efficient GPU-parallel CPG algorithm targeting CTA applications. As shown in Figure 1, the key novelty of G-PathGen is that it achieves both exact accuracy and high speed with GPU, unlike many existing algorithms that compromise one for the other. We summarize three technical contributions of G-PathGen as follows:

- We design a GPU-efficient data structure for storing a massive number of paths. On top of this structure, we introduce a criticality-aware parallel path generation algorithm that prioritizes exploring higher-criticality paths likely to be k-critical, while also considering lower-criticality paths when additional candidates are needed.
- We design a dynamic stepping strategy that carefully controls the generated path count in each stepping iteration to balance parallelism and work efficiency. The path count grows more rapidly during early stages to maximize GPU utilization and gradually slows down near completion to avoid redundant computation.
- We design GPU kernel algorithms for efficient path generation, including (1) a graph reordering algorithm that improves data locality by placing simultaneously accessed vertices close together in memory, and (2) a warp-based exploration algorithm that improves generation efficiency by assigning a group of threads to collaboratively explore each path.

We evaluate G-PathGen on large industrial circuit graphs and non-circuit graphs to demonstrate its effectiveness in both CTA applications and broader domains. Compared to G-PBA [29], G-PathGen is 1.6×–243.8× faster while producing exact results when generating one million critical paths on industrial circuit graphs. We plan to open-source G-PathGen to benefit both the CTA and parallel computing communities.

2 Preliminaries

2.1 Problem Formulation

In CTA applications, the input circuit is modeled as a *directed acyclic graph* (DAG), $G = V, E$, where V represents pins of circuit components (e.g., logic gates, flip-flops) and E represents pin-to-pin signal connections. Each directed edge e_{uv} from vertex u to v carries a weight w, representing the signal delay. A path is an ordered sequence of edges $\langle e_1, e_2, ..., e_i \rangle$, and its cost is the sum of edge weights. Figure 2 shows an example: gate pins are modeled as vertices, signal connections as edges, and pin-to-pin delays as edge weights. Given G and a positive integer k, a CPG query returns the top-k critical paths in ascending order of path cost. In practice, such graphs can contain millions of gates, and applications may query millions of critical paths [42]. Intuitively, smaller path costs represent tighter timing margins, indicating that the corresponding paths are more timing-critical and require greater attention from circuit designers.

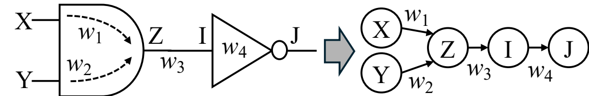

Figure 2: A circuit of two gates and its corresponding DAG.

2.2 GPU-accelerated CPG Algorithm: G-PBA

G-PBA [29] is a GPU-accelerated CPG algorithm built as part of OpenTimer [42], an open-source timer widely used by the CTA community. The key building block of G-PBA is the *implicit path representation*, which consists of two complementary data structures: a *suffix tree* and a *prefix tree*.

The suffix tree is a shortest-path tree rooted at the destination (e.g., flip-flop endpoint), serving as the basis for identifying critical path deviations. Figure 3(a) shows a DAG and its suffix tree. Black edges are part of the tree; gray edges are not. Numbers on the edge denote the weights. Numbers inside the vertices are their shortest distances to the destination (T).

The prefix tree captures alternative paths via deviations from the suffix tree. In Figure 3(b), the root Φ is the shortest path $\langle e_{SB}, e_{BE}, e_{ED}, e_{DT} \rangle$; other nodes represent deviations. Each starts with a non-suffix-tree edge, followed by suffix-tree edges. The blue node, for example, follows e_{SA}, then $\langle e_{AD}, e_{DT} \rangle$, forming the path $\langle e_{SA}, e_{AD}, e_{DT} \rangle$ (Figure 3(c)). Each prefix tree node has a *deviation cost*, which is the extra cost compared to the shortest path. In (b), the "e_{SA}" node has a cost of −5, one unit higher than the shortest path of −6, resulting in a deviation cost of +1. We refer to generating child nodes from a node as *expansion*. We use the terms "deviation cost/path cost" and "node/path" interchangeably since they are algorithmically equivalent in terms of critical path ranking. Note that the negative edge weights in Figure 3(a) arise from the timing-graph formulation used in CTA. Each edge represents either a signal delay or an offset introduced to remove clock-path pessimism [55], where the offset can be negative or positive depending on signal arrival times.

The prefix tree captures the search space of critical paths, and its expansion corresponds to exploring new path candidates. G-PBA

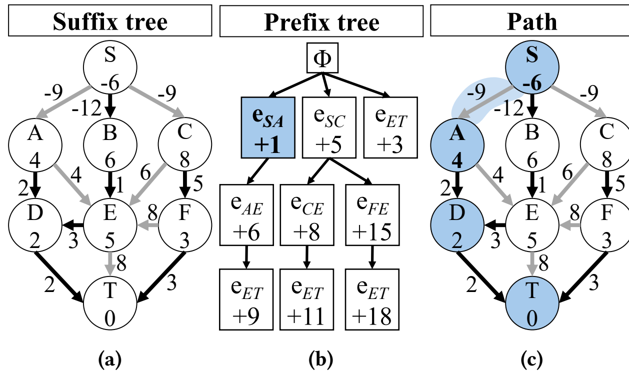

Figure 3: Implicit path representation. Prefix $\langle e_{SA} \rangle$ + Suffix $\langle e_{AD}, e_{DT} \rangle$ = Path $\langle e_{SA}, e_{AD}, e_{DT} \rangle$.

parallelizes expansion by generating each tree level concurrently on the GPU (Figure 4). At each level, GPU threads discover new paths in parallel. As shown in Figure 4(a), once the first level (e_{SA}, e_{SC}, e_{ET}) is generated, G-PBA transfers these paths to the CPU for sorting and pruning to prevent memory explosion, which discards less-critical paths (e.g., e_{SC} with deviation cost +5 is the least critical in the current level). In (b), the remaining paths (e_{SA}, e_{ET}) are sent back to the GPU to continue exploration at the next level.

As G-PBA explores deeper levels, it improves accuracy but increases runtime. This is called the *Maximum Deviation Level (MDL)* heuristic [29], which sets the depth limit to balance performance and accuracy.

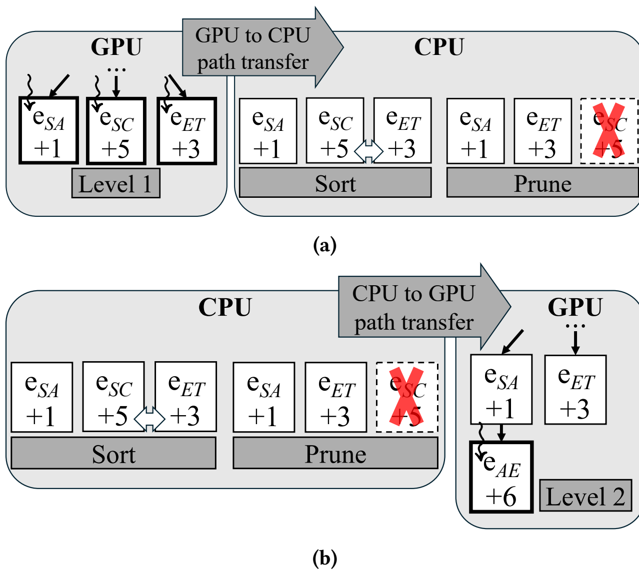

Figure 4: Illustration of the level-by-level path exploration strategy of G-PBA. (a) The GPU launches concurrently generates the level 1 of the prefix tree, and then the CPU evaluates the criticalities of each path (node) at this level. (b) After criticality evaluation and pruning, the CPU then transfers the results back to the GPU to concurrently generate level 2.

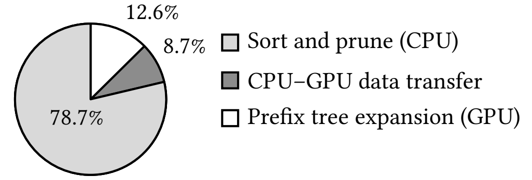

Figure 5: Runtime breakdown of G-PBA when MDL = 3 on a 4M-gate circuit (netcard). CPU–GPU data transfer and CPU-side evaluation take up the majority of the total runtime.

2.3 Limitations of G-PBA

Despite GPU-accelerated expansion, G-PBA has two major drawbacks: (1) It cannot guarantee exact k-critical paths due to the MDL heuristic. As shown in Figure 4(a) and (b), path criticalities are evaluated only within the same tree level. Thus, paths that belong to the exact k-critical set may be pruned prematurely. For example, as shown in Figure 3(b), the exact 5-critical paths should include nodes Φ, e_{SA}, e_{ET}, e_{SC}, e_{AE}, but in Figure 4(a), G-PBA considers e_{SC} to be less critical compared to other paths in the same level and discards e_{SC}. As a result, G-PBA becomes inexact. (2) It incurs high overhead from repeated CPU–GPU transfers. G-PBA offloads each tree level to the CPU for sorting and pruning, and transfers the results back. These levels can contain millions of vertices in large graphs, leading to high CPU processing and data transfer costs. For example, on a circuit with 4M gates (netcard), Figure 5 shows CPU-side sort/prune and data transfer take 78.7% and 8.7% of total runtime.

3 G-PathGen

To overcome the limitations of G-PBA, we propose G-PathGen. Inspired by the implicit path representation [42], which is widely adopted by many CTA applications [7, 8, 29], G-PathGen runs in two phases: *parallel suffix tree building* and *parallel prefix tree expansion*. We shall focus on the prefix tree expansion phase as it is the most time-consuming part that inspires the core novelty of G-PathGen.

3.1 Parallel Suffix Tree Building

This phase builds a shortest-path tree rooted at the destination [42], providing per-vertex distances for prefix tree expansion. G-PBA uses Bellman-Ford [84], launching one thread per vertex per iteration. Despite massive parallelism, many threads idle when their distances are not ready, leading to wasted computation and high scheduling overhead.

Since circuit graphs are DAGs, we instead apply topological levelization and process relaxations level by level. We implement levelization via parallel BFS [57], using shared memory to track frontiers and reduce global memory contention. The suffix tree is then constructed by level. Figure 6(a) shows an example graph with six levels (L0–L5). Figure 6(b) shows that vertices A, B, and F perform relaxations in parallel. When vertices perform relaxations on the same vertex, we use atomic operations to avoid a data race. For example, A and B atomically update the distance of S.

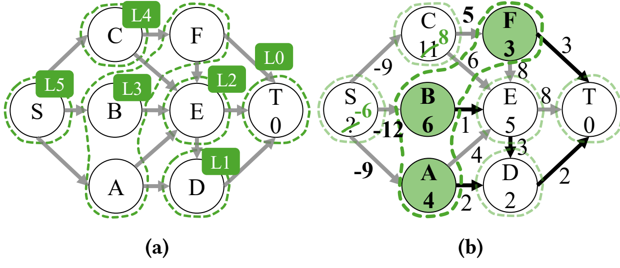

Figure 6: Illustration of (a) levelization and (b) parallel suffix tree building.

3.2 Parallel Prefix Tree Expansion

After building the suffix tree, we proceed to expand the prefix tree in parallel. The goal is to generate critical paths by identifying deviation edges along each vertex's shortest path to its destination. Unlike G-PBA, which cannot generate the exact k-critical paths due to premature pruning, we retain a minimal set of generated paths without sacrificing exactness while dynamically scheduling their expansion to avoid memory explosion. Specifically, we divide paths into low- and high-criticality groups and prioritize the expansion of high-criticality paths, as their expansion is more likely to yield k-critical paths. We defer the expansion of others, as they may become more critical in later iterations. We use a path cost threshold α to divide generated paths into two thread-safe queues: the *high-priority queue* (HPQ) for paths with cost $\leq \alpha$, and the *low-priority queue* (LPQ) for those with cost $> \alpha$. We expand paths from the HPQ until no new ones are added. If the HPQ has fewer than k paths, we increase α by step size Δ to promote additional paths from the LPQ. We refer to this design as *dual-queue (DQ) scheduling*.

Algorithm 1: Expand(*pfx*, *d*, *HPQ*, *LPQ*, α)

Input: prefix tree node *pfx*, destination vertex *d*, high-priority queue *HPQ*, low-priority queue *LPQ*, cost threshold α
Global: array of suffix tree successors *succs*

1. $u \leftarrow tail[pfx.e]$;
2. **while** $u \neq d$
3.   **Parallel Foreach** *lane* ∈ *a GPU warp*
4. $e \leftarrow$ *fanout of u* at position *lane*
5.   **if** *tail[e] == succs[u]* **then**
6.   **continue**;
7.   *pfx_new* ← new PfxtNode(*pfx*, *e*, *pfx.dv* + *dv* of *e*);
8.   *cost* ← compute path cost of *pfx_new*;
9.   **if** $cost \leq \alpha$ **then**
10.   *HPQ*.push(*pfx_new*);
11.   **else**
12.   *LPQ*.push(*pfx_new*);
13.   **synchronize all lanes** ; `// implicit barrier`
14. $u \leftarrow succs[u]$;

Algorithm 1 describes the prefix tree expansion algorithm. During expansion, this algorithm determines the priority of each node by placing it into different priority groups. We first get the tail vertex u of the deviation edge from prefix node *pfx* (line 1) and traverse the shortest path from u to destination d using the suffix tree successors (lines 2 and 13). At each vertex, unlike G-PBA, which uses one thread to scan outgoing edges, we use a GPU warp (line 3:4); All threads (lanes) in the warp simultaneously process an outgoing edge. We skip suffix-tree edges (line 5:6). For each non-suffix-tree edge, we create a new prefix node *pfx_new*, compute its path cost (line 7:8), and place it in *HPQ* if $cost \leq \alpha$ (line 9:10), or *LPQ* otherwise (line 11:12). After warp synchronization, we traverse to the suffix tree successor of u (line 14).

Algorithm 2: G-PathGen(k, P, d, *HPQ*, *LPQ*, Δ)

Input: path count k, prefix tree P, destination vertex d, high-priority queue *HPQ*, low-priority queue *LPQ*, cost threshold step size Δ
Output: k-critical path set

1. *tmp* ← initialize by expanding *P*.root;
2. sort *tmp* in ascending order of path cost;
3. *HPQ* ← top 0.5% of *tmp*;
4. *LPQ* ← bottom 99.5% of *tmp*;
5. α ← largest path cost in *HPQ*;
6. *window* ← *HPQ*;
7. **while** *true*
8.   **if** *window size* > 0 **then**
9.   **Parallel Foreach** *node* ∈ *window*
10.   Expand(*node*, *d*, *HPQ*, *LPQ*, α);
11.   **synchronize all threads** ; `// implicit barrier (kernel-level)`
12.   **else**
13.   **if** *HPQ.size()* $\geq k$ **or** *LPQ is empty* **then**
14.   **break**;
15. α += Δ;
16.   **Parallel Foreach** *node* ∈ *LPQ*
17.   *cost* ← compute path cost of *node*;
18.   **if** $cost \leq \alpha$ **then**
19.   move *node* to *HPQ* (atomically);
20.   **synchronize all threads** ; `// implicit barrier (kernel-level)`
21.   *window* ← get new nodes in *HPQ*;
22. sort *HPQ* in ascending order of path cost;
23. **return** the first k nodes of *HPQ*;

Algorithm 2 describes how G-PathGen interacts with the HPQ and LPQ to schedule the expansion of generated paths. To ensure exactness, this algorithm expands any newly added nodes in the HPQ to find all paths with costs no more than the threshold α. Unlike G-PBA, we do not prematurely prune paths. Instead, we temporarily keep less critical ones in the LPQ and promote them later if more candidates are needed.

We first initialize a temporary node array *tmp* that stores the nodes expanded from the prefix tree root (line 1) and sort it by path cost (line 2). We place the top 0.5% of *tmp* into the HPQ and the rest into the LPQ. Considering that the HPQ can grow rapidly and

may generate far more paths than needed, we suggest placing only a small percentage (e.g., 0.5%, 1%) of *tmp* into HPQ to start with.

We initialize the path cost threshold α to the largest path cost in HPQ (line 5), as this threshold separates *tmp* into two priority groups. We initialize a group of nodes *window* that is ready for expansion, which is the entire HPQ (line 6). In the main loop (line 7:21), if *window* is non-empty, indicating that we have unexpanded high-priority nodes, we launch GPU threads to expand *window* in parallel (line 9:10). Each expansion may generate new prefix nodes that are inserted into either HPQ or LPQ depending on their path costs relative to the current threshold. Here, kernel-level synchronization is invoked (line 11) to ensure that all expansions complete before proceeding. If *window* is empty, indicating that all high-priority nodes have been expanded, we check termination conditions: the algorithm stops if we have obtained $\geq k$ nodes or if *LPQ* is empty (line 13:14). If termination conditions are not met, we increase α and promote eligible *LPQ* nodes to *HPQ* (line 16:19), allowing additional candidate paths to enter the exploration frontier. Kernel-level synchronization is invoked again (line 20). We then gather the newly added *HPQ* nodes, form a new *window*, and continue the expansion (line 21). After termination, we sort *HPQ* and return the top-k paths (line 22:23).

Figure 7 illustrates Algorithm 2. For brevity, we show path costs on the nodes instead of deviation costs. In this example, we set $\alpha = -3$, $\Delta = 5$, and the path count $k = 10$; A prefix tree node that is associated with edge e_{uv} and cost W is denoted as Pfx(e_{uv}, W). We start with three nodes: Pfx(e_{SA}, −5), Pfx(e_{SC}, −1), Pfx(e_{ET}, −3). Figure 7(a) shows a suffix tree on the left and a DQ scheduler on the right. Pfx(e_{SA}, −5) and Pfx(e_{ET}, −3) enter the HPQ (cost $\leq \alpha$) and Pfx(e_{SC}, −1) enters the LPQ. These two HPQ nodes form a group (dashed box) of nodes that are assigned to two GPU threads. We refer to this node group as an *expansion window* (*window* in short). Thread 1 (orange) traverses $\langle e_{AD}, e_{DT} \rangle$ and generates a new node Pfx(e_{AE}, 0), which is outlined in bold lines. Since 0 is larger than the current threshold α, Pfx(e_{AE}, 0) enters the LPQ. Thread 2 (green) has no path to traverse since e_{ET} already reaches the destination, so it generates nothing. Since no new nodes enter the HPQ and fewer than k paths have been generated, we increase α to promote some LPQ nodes. As shown in Figure 7(b), increasing α by Δ sets $\alpha = 2$. Along with α's increase, Pfx(e_{SC}, −1) and Pfx(e_{SC}, 0) enter the HPQ. Lemma 1 highlights a key invariant that guarantees the exactness of G-PathGen.

Lemma 1. *G-PathGen only promotes nodes when α increases.*

G-PathGen maintains this invariant to properly separate high- and low-priority paths. If α is increased without promoting eligible nodes, or nodes are promoted without updating α, the DQ scheduler becomes ineffective. Now that Pfx(e_{SC}, −1) and Pfx(e_{SC}, 0) are promoted to the HPQ, they form a new window. As shown in Figure 7(c), thread 1 traverses $\langle e_{CF}, e_{FT} \rangle$ and generates two nodes: Pfx(e_{CE}, 2), which enters the HPQ, and Pfx(e_{FE}, 9), which enters the LPQ; thread 2 traverses $\langle e_{ED}, e_{DT} \rangle$ and generates Pfx(e_{ET}, 3), which also enters the LPQ. In the HPQ, Pfx(e_{CE}, 2) itself forms a new window. Lemma 2 highlights another key invariant of G-PathGen.

Lemma 2. *Once a window is formed, G-PathGen will expand all the nodes in that window until no new windows are formed.*

Together, these two invariants ensure that all paths with costs no more than α are found. As a result, once the HPQ holds at least k paths, it is guaranteed to contain the exact k-critical set. Theorem 1 proves the exactness of G-PathGen.

Theorem 1. *G-PathGen generates exact k-critical paths.*

Proof. Assume that G-PathGen fails to generate a prefix tree node n with path cost $c \leq \alpha$ that is k-critical. If n was in the HPQ, it must have participated in the final ranking process, which contradicts the assumption. If n was in the LPQ, Lemma 1 ensures that it must have been promoted to the HPQ because $c \leq \alpha$. By contradiction, Theorem 1 is correct. □

To discuss the work bound of the prefix tree expansion phase, we decompose the total cost into (1) per-node expansion work and (2) scheduler overhead. The total work is proportional to the number of generated prefix nodes and the cost of expanding each node. Expansion traverses the suffix tree successor chain from the deviation edge's tail toward the destination; at each visited vertex, a GPU warp collaboratively scans outgoing edges to enumerate non-suffix-tree deviation edges. Thus, the per-node expansion cost scales with the suffix-path length (bounded by the graph diameter) and the fanout encountered along that suffix path. In addition to expansion, the DQ scheduler adds sorting overhead: we sort the root-expanded candidates once during initialization, and we perform a single final sort of the HPQ to produce the ranked path results. The stepping rule (increasing α by Δ) controls how many LPQ nodes are promoted per step, trading off parallelism (window size) against redundant work (extra generated paths).

3.3 Dynamic Adjustment of Step Size

The step size Δ controls how fast the path cost threshold α increases, which affects the window sizes, thereby impacting the work efficiency and parallelism of G-PathGen. Since paths are generated in parallel, we want to avoid exceeding k too much, so we measure the work efficiency by the total path count in the HPQ before G-PathGen stops. We measure parallelism by the window sizes, as each window is processed by multiple GPU threads. Each update of α is one step.

Figure 8 illustrates the HPQ behavior at $\Delta = 2$ and $\Delta = 5$ with $k = 5$. $\Delta = 2$ (top) generates six paths (only one extra path), which is work-efficient, but the parallelism is limited (e.g., window size of two at $\alpha = -1$). In contrast, $\Delta = 5$ (bottom) generates eight paths, offering more parallelism (e.g., window size of five at $\alpha = 2$), but less work-efficient. Larger Δ values can cause G-PathGen to generate many non-k-critical paths. Our experiments show that with a larger Δ, the total number of generated paths can exceed twice the k value for large circuits, leading to very low work efficiency. To overcome this challenge, we introduce a dynamic stepping strategy that (1) starts with an adequately small Δ to avoid generating too many non-k-critical paths at early stepping iterations and (2) dynamically adjusts Δ to control the generated path count per step while balancing work efficiency and parallelism.

Based on sampled path costs from our graph set, we observe that the choice of initial Δ depends heavily on graph density. For example, $\Delta = 40$ is too large for dense graphs, since they tend to have closely packed path costs (e.g., 0.1, 0.13, 0.19); it creates

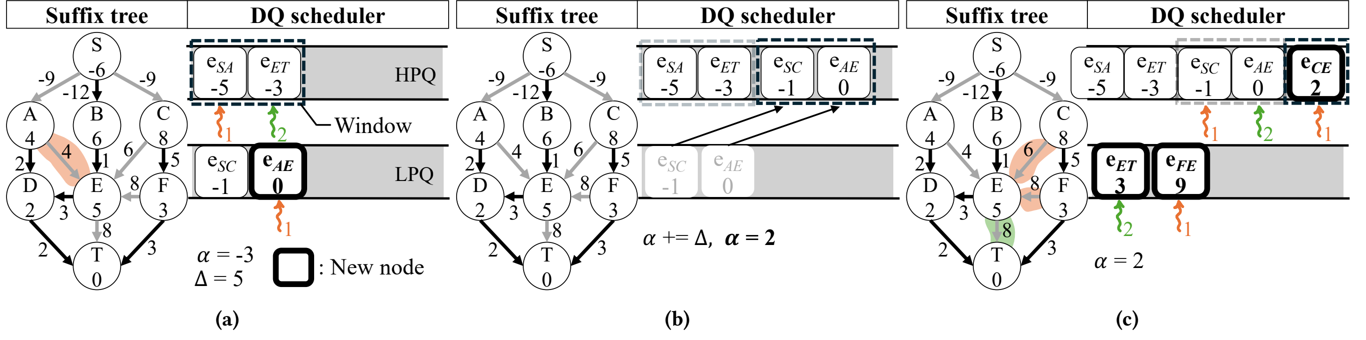

Figure 7: Illustration of parallel prefix tree expansion. Dashed lines group nodes in a window; bold outlines indicate new nodes. We set $\alpha = -3$, $\Delta = 5$, and $k = 10$. (a) Pfx(e_{SA}, -5) and Pfx(e_{ET}, -3) enter the HPQ (cost ≤ -3), forming the initial window. Pfx(e_{SC}, -1) enter the LPQ. Expanding Pfx(e_{SA}, -5) yields Pfx(e_{AE}, 0), which enters the LPQ. (b) HPQ has no new windows, so we increase α by Δ, promoting Pfx(e_{SC}, -1) and Pfx(e_{AE}, 0) into HPQ. (c) Thread 1 expands Pfx(e_{SC}, -1), generating Pfx(e_{CE}, 2) and Pfx(e_{FE}, 9); thread 2 expands Pfx(e_{AE}, 0), generating Pfx(e_{ET}, 3). Pfx(e_{CE}, 2) enters HPQ and forms a new window.

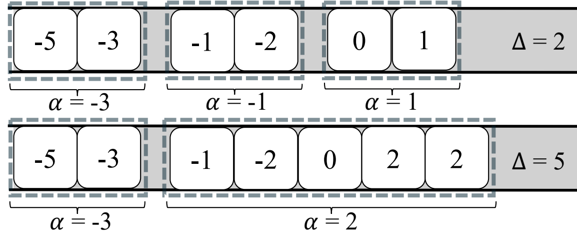

Figure 8: Illustration of work efficiency and parallelism of G-PathGen. The label under each window (dashed lines) indicates that the window was formed at $\alpha = N$.

excessively large windows and may generate too many non-k-critical paths. In contrast, $\Delta = 40$ is adequately small for sparse graphs, since they tend to have widely spaced path costs (e.g., 10, 15, 27); it creates windows of moderate sizes and can prioritize paths of higher criticality. Thus, we define a sigmoid function to determine the initial Δ:

$$\Delta_{init}(d) = \Delta_{min} + \frac{\Delta_{max} - \Delta_{min}}{1 + e^{s(d-d_0)}} \tag{1}$$

We measure the graph density d by the average vertex degree of that graph. Based on sampled graphs, the initial step size Δ_{init} is bounded by $\Delta_{min} = 0.1$ and $\Delta_{max} = 100$, with an inflection point at $d_0 = 3$. This is the inflection point where the sigmoid curve bends. We set the steepness $s = 0.8$ so that Δ_{init} drops more quickly as graph density increases. This avoids excessive path generation on dense graphs, improving work efficiency. Although this increases the number of steps on sparse graphs, our experiments show that the runtime increase is minimal.

To dynamically adjust Δ during execution, G-PathGen checks the HPQ whenever no new paths are added. If the HPQ has fewer than $k/2$ paths, which indicates slow progress, we multiply Δ by a tunable parameter m (default 1.2) to enlarge window sizes and boost parallelism. Once the HPQ exceeds $k/2$ paths, we reset Δ to Δ_{init} to slow down path generation and avoid generating too many non-k-critical paths.

3.4 Graph Reordering

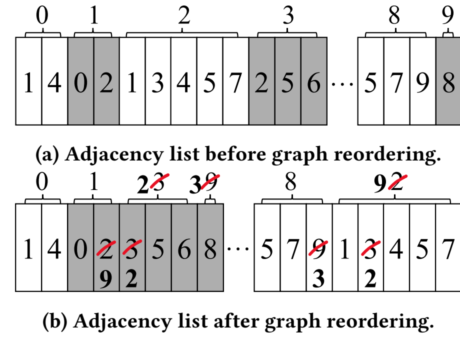

Figure 9: Illustration of graph reordering. Gray adjacency lists are accessed together. (a) Before reordering, the adjacency lists of vertices 1, 3, and 9 are scattered in memory, resulting in poor data locality when they are accessed together. (b) After reordering, the accessed vertices are relabeled and placed close to each other (vertices 1, 2, and 3), so their adjacency lists are contiguous in memory, improving data locality.

Graph reordering [1, 3, 12, 65, 71, 97, 101] is a widely used technique that rearranges the vertex layout based on their memory access pattern to improve data locality. Figure 9 illustrates an example of graph reordering. In Figure 9(a), the adjacency lists of vertices 1, 2, and 3 are accessed together, but they are far apart, resulting in poor memory locality. In (b), after reordering and relabeling vertex IDs, the adjacency lists of the accessed vertices are placed close together, improving data locality.

To further enhance the performance of G-PathGen, we reorder the input graph based on the vertex access pattern to improve data locality. Although graph reordering algorithms have been extensively studied, we found two major challenges when applying them to our problem: (1) Existing algorithms are designed for general-purpose graph applications and do not fully exploit the algorithmic

structure of G-PathGen. (2) In addition, they treat graph reordering as a one-time CPU preprocessing step. However, for large CPG problems, this reordering time can become a bottleneck, as we shall demonstrate in the later experiment.

Instead of designing yet another vertex reordering algorithm, we utilize the levelization result from our suffix tree building phase to assign contiguous IDs to same-level vertices. Since both suffix tree building and prefix tree expansion access vertices level by level, this order improves data locality. To minimize the reordering time, we introduce a GPU-parallel algorithm (Algorithm 3) to efficiently update the graph's Compressed Sparse Row (CSR) data structure. We update vertex positions directly on the GPU without CPU involvement. CSR consists of an adjacency pointer array, an adjacency list, and a weight array; we omit the weight array here, as it is updated similarly to the adjacency list. We update the adjacency pointers in advance using a GPU-parallel prefix scan kernel [87].

Algorithm 3: UpdateCSR(*new_ids*, *vs*, *es*, *inv_es*, *u_vs*)

Input: new index mapping *new_ids*, adjacency pointer *vs*, adjacency list *es*, edge-to-head map *inv_es*, updated adjacency pointer *u_vs*

Global: number of edges *M*, updated adjacency list *u_es* (initially empty)

```
1  Parallel Foreach thread
2    tid ← blockIdx.x × blockDim.x + threadIdx.x;
3    for i ← tid to M by grid_stride do
4      u ← inv_es[i];
5      v ← es[i];
6      offset ← i − vs[i];
7      new_u ← new_ids[u];
8      new_e_beg ← u_vs[new_u];
9      new_v ← new_ids[v];
10     u_es[new_e_beg + offset] ← new_v;
```

In Algorithm 3, we summarize the arrays required for the CSR update. *vs* is the adjacency-pointer array, *es* is the adjacency list, and *inv_es* is an inverse map of *es* constructed during graph input (mapping each edge index to its head vertex). Using *inv_es*, given an edge index, the algorithm retrieves the corresponding head vertex so it can update that edge's position in the CSR layout. Each thread first computes its global index *tid* (line 1) and then uses a grid-stride loop to process multiple edges when M exceeds the total thread count (line 3:10). For each edge index, we use *inv_es* to obtain the head vertex u (line 4) and read the tail vertex v from *es*. We then compute the offset of this edge within u's neighbor list (line 6), the new index of u (line 7), and the starting position for writing u's neighbors in the updated adjacency list (line 8). Finally, we compute the new index of v and write it to the updated adjacency list (line 10).

4 Experimental Results

We implemented G-PathGen in CUDA 12.6 and compiled it with nvcc on a host compiler of GCC-11.4. We enabled the optimization flag `-O3` and C++20 standard `-std=c++20`. We ran experiments on a 64-bit Linux machine with 20 Intel i5-13500 CPU cores at 4.8 GHz and an Nvidia RTX A4000 GPU of 16 GB RAM. Since G-PathGen is inspired by CTA applications, we evaluated G-PathGen on six large industrial circuit graphs, generated by an open-source CTA tool, OpenTimer [42]. To further evaluate the performance of G-PathGen on non-circuit graphs, we select four large non-circuit graphs from DIMACS Graph Challenge [2]. For non-DAG graphs, we induced a direction for each edge from smaller to larger vertex IDs, thus avoiding cycles. Table 1 lists the graph statistics. By default, G-PathGen enables warp-based prefix tree expansion, dynamic Δ adjustment, and graph reordering for best performance. We validate the path costs generated by G-PathGen by comparing them against the exact path costs generated by OpenTimer. We define the average path cost error (Err_{avg}) as the average relative difference across all reported paths. We define the maximum path cost error (Err_{max}) as the maximum relative difference among all reported paths.

4.1 Baseline Algorithm

We consider G-PBA [29], a state-of-the-art GPU-accelerated k-critical path generator targeting CTA applications, as our baseline. We compare only against G-PBA because it has been shown to be the most efficient GPU-parallel CPG algorithm in both space and time and is widely used in many CTA applications [8, 42]. Moreover, G-PBA is the only GPU-parallel CPG algorithm based on an implicit path representation, where the prefix-tree structure enables efficient storage of millions of critical paths required by CTA applications. In practice, this prefix-tree-based exploration has been observed to outperform several CPU-parallel k-CPG algorithms (e.g., Yen and OptYen), making G-PBA a strong and representative baseline for evaluating GPU-parallel CPG methods.

Despite GPU acceleration, G-PBA is not exact because it relies on a heuristic called *Maximum Deviation Level (MDL)* to prune paths, avoiding memory explosion. Specifically, MDL controls the maximum depth that G-PBA can explore and can be tuned to balance performance and accuracy. A higher MDL enables deeper exploration, potentially improving accuracy at the cost of increased runtime, and vice versa. Additionally, to avoid exceeding the GPU memory limit, G-PBA offloads the discovered paths to the CPU at each level of the search. The CPU sorts and prunes these paths, retaining only the more critical ones. The selected paths are then transferred back to the GPU to continue the search. For brevity, we refer to G-PBA with an MDL of N as $\text{G-PBA}_{L=N}$.

4.2 Overall Performance Comparison

Table 1 compares the overall performance and accuracy between G-PBA and G-PathGen. It also lists three important graph statistics: number of vertices (|V|), number of edges (|E|), and graph diameter (|D|). Graph diameter is the longest distance between any two vertices. Since practical CTA applications often need to analyze millions of critical paths [42], we set the path count k to 1M. To compare accuracy, we show the average/maximum path cost error (Err_{avg}/Err_{max}) across 1M paths. To compare performance, we show the runtime for both suffix tree building (Sfxt) and prefix tree expansion (Pfxt). The total elapsed runtime is the sum of both.

In terms of accuracy, G-PathGen always produces *exact* path results. Its Err_{avg} and Err_{max} are always 0%. This is because it

Graph	|V|	|E|	|D|	G-PBA [29] Err$_{avg}$ (%)	Err$_{max}$ (%)	Sfxt (ms)	Pfxt (ms)	Total (ms)	G-PathGen Err$_{avg}$ (%)	Err$_{max}$ (%)	Sfxt (ms)	Pfxt (ms)	Total (ms)
c7522	3.8K	152K	71	0.408	3.488	3.9	24260.2	24264.1	0.000	0.000	2.7 (1.5×)	96.9 (250.4×)	99.5 (243.8×)
des_perf	303.6K	12.1M	201	1.693	25.464	267.1	720	987.1	0.000	0.000	23.3 (11.5×)	82.9 (8.7×)	106.2 (9.3×)
vga_lcd	397.8K	15.9M	212	8.741	11.988	364.7	817.9	1182.5	0.000	0.000	23.9 (15.3×)	714.7 (1.1×)	738.6 (1.6×)
leon3mp	3.3M	135M	482	20.354	27.531	12678.9	19196.1	31875	0.000	0.000	107.7 (117.7×)	168.0 (114.3×)	275.7 (115.6×)
netcard	3.9M	159.9M	480	18.004	26.442	17358	225.3	17583.2	0.000	0.000	130.4 (133.1×)	4615.6 (0.05×)	4746.0 (3.7×)
leon2	4.3M	173.1M	503	41.641	53.98	21036.6	238.3	21275	0.000	0.000	146.8 (143.3×)	253.6 (0.9×)	400.4 (53.1×)
ldoor	952.2K	22.7M	6734	1.777	4.079	601.6	29752.4	30354	0.000	0.000	234.6 (2.6×)	188.5 (157.9×)	423.1 (71.7×)
cage15	5.1M	47M	147	0.854	1.391	131	15566.4	15697.4	0.000	0.000	32.4 (4.0×)	90.3 (172.4×)	122.7 (127.9×)
nlpkkt120	3.5M	46.6M	2	0	0.01	7.9	3376.3	3384.2	0.000	0.000	9.8 (0.8×)	211.8 (15.9×)	221.6 (15.3×)
nlpkkt160	8.3M	110.5M	2	0	0.01	18.8	8166.3	8185	0.000	0.000	23.0 (0.8×)	459.8 (17.8×)	482.8 (17.0×)

|D|: graph diameter. Err$_{avg/max}$: average/max path cost error. Sfxt/Pfxt: suffix tree building/prefix tree expansion runtime.

Table 1: Overall performance and accuracy comparison between G-PBA and G-PathGen. G-PathGen achieves exact accuracy and the fastest total runtime across all graphs.

computes a threshold that filters out non-k-critical paths. In contrast, G-PBA is not exact. For example, its Err$_{avg}$ reaches 20.3% on leon3mp and 41.6% on leon2. This is because it uses the MDL heuristic to avoid memory explosion, but it may prune k-critical paths and become inaccurate.

In terms of performance, G-PathGen outperforms G-PBA in Sfxt, Pfxt, or both on most graphs. For example, in Sfxt, G-PathGen is 133.1× and 143.3× faster than G-PBA on netcard and leon2, respectively. This is because G-PathGen's levelizes the graph first and visits each vertex exactly once. In contrast, G-PBA visits each vertex multiple times, leading to many unnecessary updates. In Pfxt, for example, G-PathGen is 250.4× and 172.4× faster than G-PBA on c7522 and cage15, respectively. This is because G-PathGen runs mostly on the GPU with minimal CPU involvement, while G-PBA repeatedly transfer paths between the CPU and GPU to prune paths, leading to significant data transfer overhead. G-PathGen's Pfxt is slower on some graphs. For example, G-PathGen's Pfxt is 20× slower on netcard. This is because G-PBA explores only a few levels of the path search space before hitting memory limits, resulting in a lower runtime. However, the generated paths by G-PBA are far from accurate. For example, on netcard, G-PBA's Err$_{avg}$ and Err$_{max}$ are 18% and 26.4%, respectively.

4.3 Analysis of Performance and Accuracy under Different Path Counts

Figure 10 plots the runtime and average path cost error (Err$_{avg}$) at different path counts (k) on one circuit and one non-circuit graph, vga_lcd and ldoor. We only plot G-PBA$_{L=2}$ and G-PBA$_{L=4}$ because G-PBA is unable to explore beyond an MDL of four due to memory constraints. In terms of performance, G-PBA's runtime remains nearly constant. For example, on vga_lcd, regardless of k, G-PBA$_{L=2}$ and G-PBA$_{L=4}$ take about 1.1 and 1.2 s; on ldoor, G-PBA$_{L=2}$ and G-PBA$_{L=4}$ take about 1.2 and 5.7 s. This is because G-PBA always explores a fixed number of levels and paths in the search space. On the other hand, G-PathGen's runtime increases along with k. For example, its runtime is about 36 and 200 ms when k = 100 and k = 100K. This is because it generates only the necessary paths

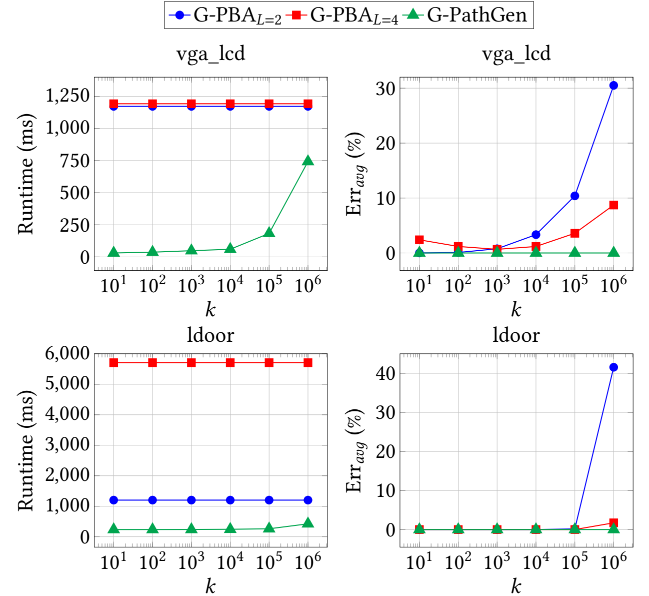

Figure 10: Runtime and average path cost error (Err$_{avg}$) under different path counts (k) on vga_lcd and ldoor. Regardless of the path count, G-PathGen always has no error and the fastest runtime.

to get the exact solution. As will be discussed in Section 4.6, G-PathGen generates only slightly more than k paths, minimizing its workload.

In terms of accuracy, G-PBA's average error generally increases with k. For example, on vga_lcd, G-PBA$_{L=2}$'s average error increases from 0.7% to 10.3% as k increases from 1K to 100K; on ldoor, it increases from 0.1% to 41.5% as k increases from 100K to 1M. This is because, as k increases, the k-critical paths tend to appear deeper in the search space, which G-PBA's limited exploration often misses. When k is small, G-PBA$_{L=2}$ is more accurate than G-PBA$_{L=4}$. For example, on vga_lcd, when k = 10, G-PBA$_{L=2}$ correctly generates all paths, while G-PBA$_{L=4}$ has an average error of 2.4%. This is

because, as G-PBA explores more of the search space, it may prune an increasing number of k-critical paths. However, when k is large, G-PBA$_{L=4}$ is more accurate. For example, at k = 1M on ldoor, G-PBA$_{L=2}$ has an average error of 41.5% vs. 1.7% for G-PBA$_{L=4}$. This is because a higher MDL preserves more late-discovered paths, which improves accuracy at larger k. In contrast, G-PathGen is exact with 0% average error, regardless of k. This is because G-PathGen computes a threshold that filters out non-k-critical paths.

4.4 Analysis of Performance and Accuracy under Different Graph Densities

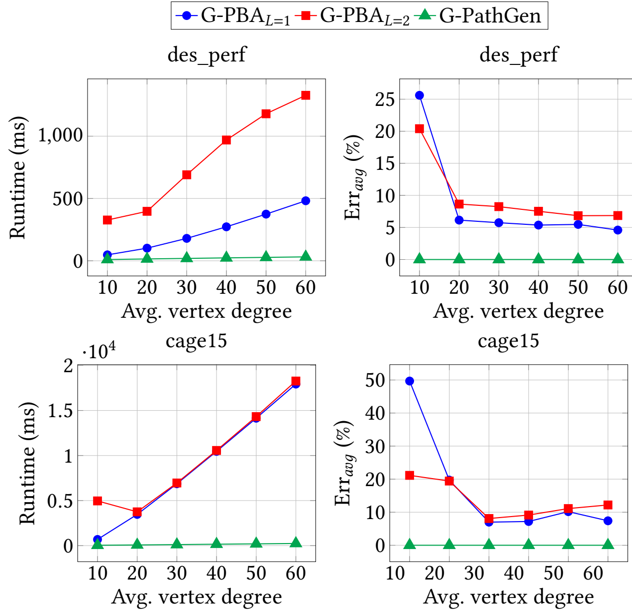

Figure 11: Runtime and average path cost error (Err$_{avg}$) under different graph densities on des_perf and cage15. Regardless of the graph density, G-PathGen is always faster than G-PBA with no error.

Figure 11 plots the runtime and average path cost error (Err$_{avg}$) at different graph densities in terms of the average vertex degree. We select one circuit graph and one non-circuit graph, des_perf and cage15. We generate graphs of different densities by randomly inserting edges into the selected graphs. We only plot G-PBA$_{L=1}$ and G-PBA$_{L=2}$ because G-PBA is unable to explore beyond an MDL of two due to memory constraints.

In terms of performance on des_perf, G-PBA$_{L=2}$'s runtime grows much faster than G-PBA$_{L=1}$ as the graph becomes denser. For example, as the graph density increases from 20 to 60, G-PBA$_{L=2}$'s runtime increases from about 397 to 1327 ms, while G-PBA$_{L=1}$'s runtime increases from about 101 to 481 ms. This is because G-PBA frequently sends path results back to the CPU for pruning to stay within the GPU memory limit. Higher graph density results in the pruning of more paths, leading to more CPU–GPU data movement overhead. In terms of performance on cage15, where the graph densities range from 20–60, G-PBA$_{L=2}$'s runtime is just slightly higher than G-PBA$_{L=1}$'s runtime. For example, at a density of 50, G-PBA$_{L=1}$'s runtime is about 14.1 s, while G-PBA$_{L=2}$'s is about 14.3 s, which is only 1% higher. Since G-PBA$_{L=2}$ only finds very few paths at MDL = 2, it processes just slightly more paths than G-PBA$_{L=1}$ and runs nearly as fast. In contrast, G-PathGen avoids repeated pruning by operating only on high-priority paths and deferring low-priority ones in memory. Therefore, it achieves lower runtime than G-PBA. For example, at a density of 60 on cage15, G-PathGen is 75× and 78× faster than G-PBA$_{L=1}$ and G-PBA$_{L=2}$, respectively.

In terms of accuracy, as the graph density increases, G-PBA's error drops and then stops improving as much. For example, on des_perf, when the graph density increases from 10 to 60, G-PBA$_{L=1}$'s error drops from around 25% to 8%, and G-PBA$_{L=2}$'s error drops from around 20% to 5%; we see similar trends on cage15; G-PBA$_{L=1}$'s error drops from around 50% to 9%; G-PBA$_{L=2}$'s error drops from around 20% to 9%. This is because denser graphs have more path candidates, so G-PBA is more likely to retain the k-critical ones. In contrast, G-PathGen always yields no error, as it computes a threshold that filters out non-k-critical paths.

4.5 Analysis of Performance and Accuracy under Different Graph Diameters

|D|	<40	40–80	80–160	160–320	>=320
# graphs	2 (2%)	13 (15%)	20 (23%)	31 (36%)	20 (23%)

Table 2: Distribution of graph diameters across 86 real-world graphs. This distribution shows that the evaluated graphs cover a wide range of graph diameters.

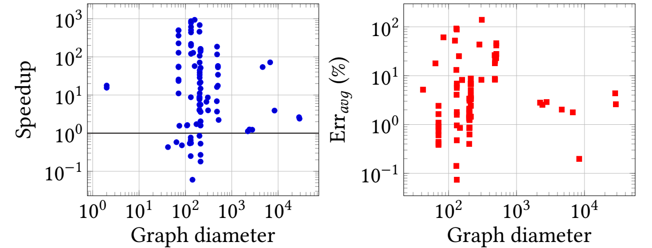

Figure 12: Speedup of G-PathGen over G-PBA and Err$_{avg}$ of G-PBA under different graph diameters. The solid black line in the speedup plot represents a speedup of 1×. G-PathGen is faster than G-PBA for more than 88% of the evaluated graphs with no error.

To further study G-PathGen's speedup over G-PBA and G-PBA's accuracy under different graph diameters, we select 86 real-world graphs from OpenTimer [42] and DIMACS [2], including those from Table 1. These graphs cover a broad diameter range as shown in Table 2, highlighting the strengths of G-PathGen across different graph structures.

Figure 12 plots the speedup and G-PBA's average path cost error (Err$_{avg}$) across different graph diameters. We omit Err$_{avg}$ of G-PathGen, as it is exact. In the speedup plot, G-PathGen outperforms

G-PBA on 76 out of 86 (> 88%) graphs. This is because, unlike G-PBA, which incurs costly CPU–GPU path transfers, G-PathGen runs mostly on the GPU with minimal CPU involvement. On a few graphs with diameters between 100–200, G-PBA's is faster because it explores very few paths and stops early. However, G-PBA's insufficient exploration often leads to major accuracy loss. For example, on tv80 (one of the circuit graphs in OpenTimer's benchmark set), G-PBA is 16.6× faster but incurs an average error of 25.1%.

4.6 Work Efficiency Analysis under Different Step Sizes

We study the work efficiency of G-PathGen under different step sizes (Δ) on one circuit and one non-circuit graph, netcard and cage15. The work efficiency is measured in terms of the total number of generated paths before the algorithm stops. G-PathGen explores paths while ensuring that whenever the cost threshold α is updated, all the paths with costs $\leq \alpha$ are found. It then increases α by a step size Δ to explore more. Each update of α is one step. Similar to Table 1, we set the path count $k = 1$M.

The choice of Δ is critical, as it affects G-PathGen's performance by affecting both the parallelism at each step and the total workload. A small Δ reduces the total generated path count, but also reduces the number of concurrently processed paths. A large Δ increases concurrency, but also increases the total generated path count. We only study the performance of Pfxt since the Δ values do not affect the performance of Sfxt. We use small Δ values (2–8 for cage15 and 20–80 for netcard) to better observe how the step count and workload vary, as larger Δ values group many similar-cost paths into the same step, leading to a massive generated path count. For brevity, we denote G-PathGen with $\Delta = d$ as G-PathGen$_{\Delta=d}$.

Figure 13(a) plots the Pfxt runtime and the step count of G-PathGen at different Δ on cage15. In (a), increasing Δ from two to four reduces the runtime from 85.7 to 68.4 ms. To explain this improvement, (b) analyzes the workload and step count on G-PathGen$_{\Delta=2}$ and G-PathGen$_{\Delta=4}$. The $\Delta = 2$ curve shows that G-PathGen generates about 1.2M paths to find the exact 1M-critical paths. This is because, although G-PathGen computes a threshold to filter out irrelevant paths, the threshold is not perfectly tight. The $\Delta = 4$ curve shows G-PathGen also stops at 1.2M paths. Since G-PathGen$_{\Delta=4}$ executes fewer steps, it runs faster. However, increasing Δ does not always improve performance. For example, in (a), increasing Δ from four to six increases the runtime from 68.4 to 108.5 ms. In (b), the curve of $\Delta = 6$ shows that G-PathGen stops after generating about 2.7M paths due to a larger computed threshold, leading to higher runtime.

Unlike fixed Δ, the dynamic strategy strikes a good balance between speed and total workload. In (a), the Δ = dyn bar shows that G-PathGen$_{\Delta=dyn}$ is 18.2 ms faster (16.8% faster) than the slowest fixed setting at $\Delta = 6$. In (b), the Δ = dyn curve shows that G-PathGen stops at about 1.5M paths, which is only slightly more than the best case of 1.2M paths and far fewer than the 2.7M paths at $\Delta = 6$. The advantage of the dynamic strategy is even more evident on the other circuit, netcard: in (d), the Δ = dyn bar shows that G-PathGen$_{\Delta=dyn}$ stops at about 1.1M paths and is 16 ms faster (27.2% faster) than the slowest fixed setting at $\Delta = 20$. This is because G-PathGen$_{\Delta=dyn}$ carefully adjusts Δ during execution: it increases Δ when progress is slow and reduces it near completion, avoiding excessive path generation while maintaining speed.

4.7 Effectiveness of Warp-based Expansion

We study the impact of warp-based prefix tree expansion (W-Pfxt) on the performance of G-PathGen. W-Pfxt assigns an entire GPU warp to traverse and find path candidates along a path (i.e., expand), rather than a single thread. Since each path candidate along each traversal is independent, W-Pfxt concurrently processes multiple path candidates from a single expansion, which improves search performance.

Figure 14 compares Pfxt runtime with and without W-Pfxt. W-Pfxt achieves speedup on all the graphs. For example, it provides a 21% speedup on leon3mp (215 vs. 168 ms) and 14% on leon2 (296 vs. 253 ms). The performance gain is small on some graphs. For example, the speedup is only 1% on netcard (4537 vs. 4492 ms). This is because W-Pfxt increases thread count and memory traffic, which can offset its benefits.

4.8 Effectiveness and Efficiency of Graph Reordering

We compare our GPU-parallel reordering algorithm with Rabbit [1], Gorder [97], and Corder [12], which represent mainstream categories of *community-based*, *neighbor-based*, and *degree-based* approaches. We select the two largest circuit and non-circuit graphs. Note that since Gorder and Corder do not support weighted graphs, we convert our graphs to unit weight in this experiment.

Figure 15 plots the runtime breakdown of G-PathGen with different reordering algorithms. The stacked bar segments (from bottom to top) plot the runtimes of three phases in G-PathGen: levelization, Sfxt, and Pfxt, respectively. Reordering improves Sfxt on some graphs. For example, on netcard, when compared to the original graph, the runtime improvement is 6.7% with Rabbit, 27.1% with Gorder, 13.6% with Corder, and 40.4% with our method. Our method and Gorder outperform Rabbit and Corder by better aligning with the level-by-level memory access of Sfxt. Specifically, our method places same-level vertices adjacent in memory, which directly matches the access pattern of Sfxt. Gorder does not target levelized structures but instead clusters vertices with shared incoming neighbors, which helps but less directly. Rabbit and Corder focus on community and workload balancing, which poorly match Sfxt's access pattern, resulting in minimal gain.

Pfxt also benefits from reordering on some graphs. For example, on leon2, the runtime improvement is 5.3% with Rabbit, 75.1% with Gorder, 20.6% with Corder, and 12.3% with our method. Gorder achieves the most improvement as it clusters vertices with common neighbors, which improves locality during edge traversal. Corder and our method achieve moderate improvements by aligning with some common memory access behaviors observed in large graphs; Corder places the high-degree vertices close together in memory, which benefits Pfxt when many paths contain such vertices; our method places same-level vertices adjacent in memory, which benefits Pfxt when many threads simultaneously access the same level. Rabbit offers minimal improvement as Pfxt's access pattern does not follow community-like structures.

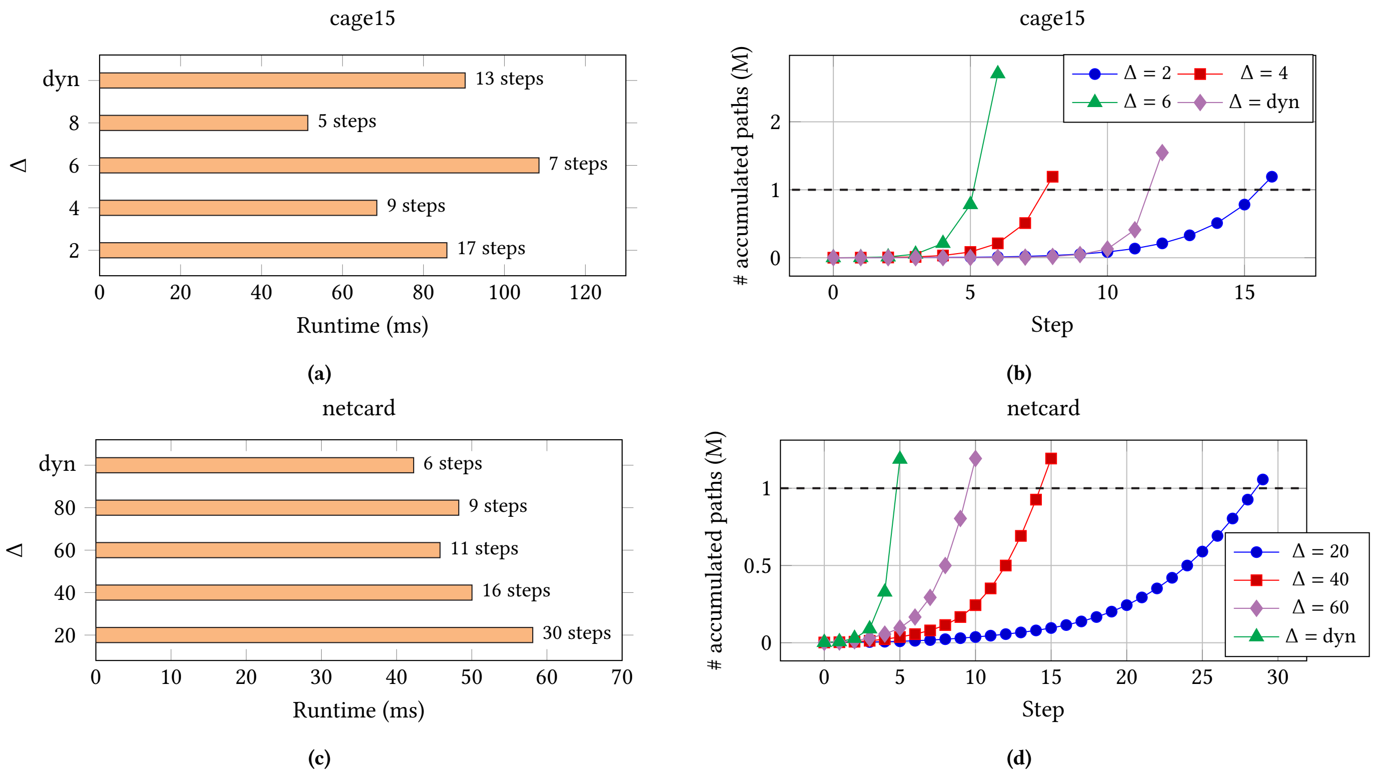

Figure 13: (a)/(c) Prefix tree expansion runtime and step count under fixed and dynamic step sizes (Δ) on cage15/netcard. (b)/(d) Accumulated path count per step under different step sizes on cage15/netcard. The black dashed line represents k = 1M. The dynamic stepping strategy achieves a great balance between speed and work efficiency; it always runs faster than the slowest fixed Δ setting, and the generated path count does not exceed k too much.

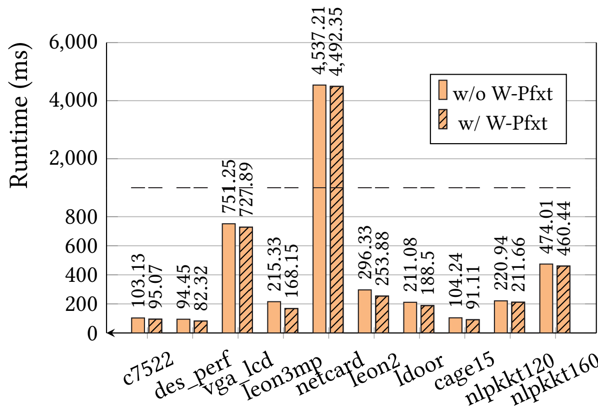

Figure 14: Prefix tree expansion runtime w/ and w/o W-Pfxt. Assigning a warp to each expansion improves performance by up to 21% compared to the original approach.

Although Rabbit, Corder, and Gorder improve the performance of G-PathGen, their reordering time becomes significant on large graphs and can outweigh the performance gains. Since they are CPU-based, their reordering time consists of (1) computing a vertex order and updating the CSR on the CPU, and (2) transferring the updated CSR to the GPU. In contrast, our algorithm performs only a GPU-parallel CSR update without CPU involvement (Algorithm 3).

Figure 16 shows that the reordering time of Rabbit, Gorder, and Corder outweighs their advantages on most graphs. Take leon2 for example, when comparing Figure 16 with Figure 15, Rabbit, Gorder, and Corder improve G-PathGen by 4.2 ms, 86.1 ms, and 24.9 ms, respectively, but require 5.1 s, 31.3 s, and 489 ms to reorder. This is because these algorithms require vertex permutations or clustering to explore data locality, which run slowly on large graphs. In contrast, our method yields a net performance gain on most graphs, as the reordering time is much faster yet effective. For example, on leon2, our algorithm reduces runtime by 27.8 ms while requiring only 19 ms to reorder, resulting in a net gain of 8.8 ms. This is because our algorithm leverages the levelization results as the new vertex order and updates the CSR directly on the GPU, which avoids both costly vertex permutations and data transfers.

5 Related Work

To quickly generate critical paths, the CTA community has proposed several efficient single-threaded algorithms. iTimerC [66] proposed branch-and-bound to prune irrelevant paths. iitRACE [90]

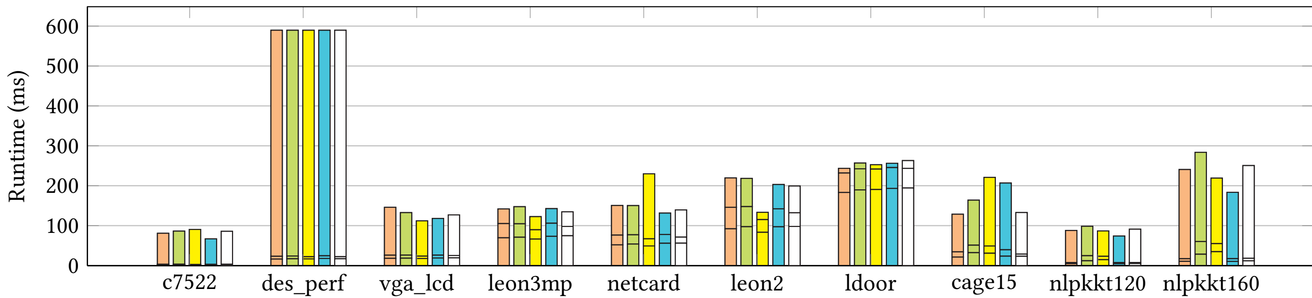

Figure 15: G-PathGen runtime breakdown with different graph reordering algorithms (Original, Rabbit, Gorder, Corder, Ours). Stacked bar segments (from bottom to top) plot the runtimes of levelization, suffix tree building, and prefix tree expansion, respectively. Graph reordering improves the performance of both suffix tree building and prefix tree expansion.

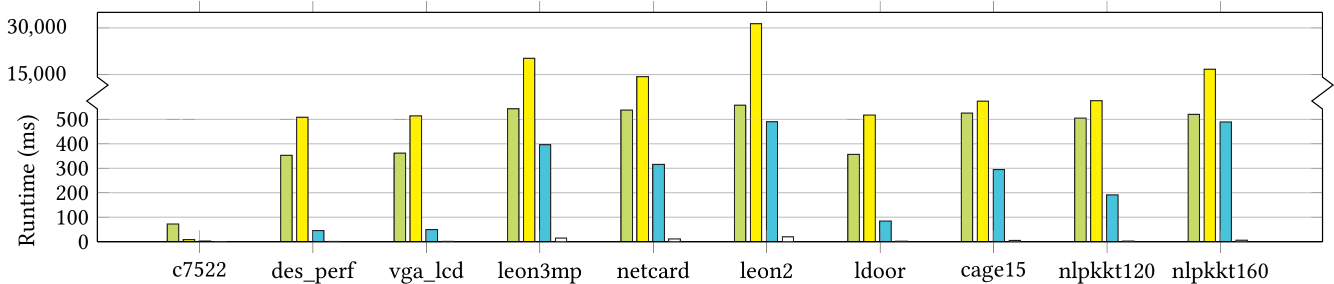

Figure 16: Graph reordering time of different algorithms (Rabbit, Gorder, Corder, Ours). Our reordering time only consists of CSR update on the GPU without CPU involvement.

proposed pin coloring to focus on affected timing regions. OpenTimer [42] proposed an efficient path representation for fast exploration. Multithreaded algorithms have been introduced to further enhance CPG performance. Ruppert [91] proposed parallel tree contraction to extract each path from its corresponding representation concurrently. PeeK [24] proposed hierarchical parallelism by distributing shortest-path tasks across computing nodes and cores. PathGen [8] proposed multi-level queue scheduling to explore paths of similar priority in parallel. However, these algorithms are limited to CPU parallelism and remain slow for large CTA problems.

To address this limitation, researchers proposed GPU-accelerated algorithms to explore data parallelism during the path generation process. Singh et al. [94] proposed GPU-parallel shortest-path tree computation but did not parallelize the most time-consuming path search. G-PBA [29] maps exploration tasks to GPU threads but lacks exactness, which limits its adoption in CTA applications. Beyond these efforts, most state-of-the-art graph processing frameworks (e.g., GBBS [21], GAPBS [4], PASGAL [22], and Gunrock [88]) are primarily CPU-oriented and/or focus on Single-Source Shortest Path (SSSP) workloads, and therefore do not directly support GPU-parallel k-shortest or k-critical path generation. As a result, GPU-parallel algorithms for k-shortest or k-critical path generation remain scarce.

Regarding graph reordering, mainstream approaches are categorized into *degree-based* [3, 12, 101], *community-based* [1, 71], and *neighbor-based* [65, 97]. Degree-based algorithms like HC [3] and FBC [101] cluster high-degree (*hot*) vertices to increase cache utilization. To mitigate load imbalance when grouping hot vertices, Corder [12] balances the number of edges across partitions. Community-based algorithms assign contiguous IDs to the vertices of the same community to improve data locality. Rabbit [1] detects communities that map to cache hierarchies to increase hit rate. SlashBurn [71] decomposes communities to reveal locality structures. Neighbor-based algorithms evaluate locality based on the shared incoming neighbors between vertices. Gorder [97] and ReCALL [65] quantify temporal and spatial locality based on shared neighbors and perform iterative optimization. While these reordering algorithms can improve locality, they target general-purpose graph applications instead of leveraging G-PathGen's algorithmic structure. Also, their CPU-based execution can become a bottleneck for large CPG problems.

6 Conclusion

We have presented G-PathGen, an efficient and exact GPU-parallel CPG algorithm targeting CTA applications. G-PathGen introduces efficient kernel algorithms for generating critical paths in parallel and dynamically adjusts the generated path count to maximize GPU utilization while minimizing redundant work. Compared to a state-of-the-art GPU solution, G-PathGen is 1.6×–243.8× faster when generating one million critical paths on industrial circuit graphs. We plan to integrate G-PathGen into the open-source timing analysis took, OpenTimer [42], to facilitate more interdisciplinary research

between the CTA and HPC communities. Inspired by the success of existing CTA research in leveraging task graph parallelism [6–11, 13–20, 23, 25–36, 38–54, 56, 58–64, 67–70, 72–83, 85, 86, 92, 93, 95, 96, 98–100, 102], our future work is to further accelerate G-PathGen using CUDA Graph.

Acknowledgments

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. The authors would like to also thank reviewers for their constructive feedback in improving this paper.

References

[1] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 22–31. doi:10.1109/IPDPS.2016.110
[2] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop*. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science.
[3] Vignesh Balaji and Brandon Lucia. 2018. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 203–214.
[4] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619
[5] Jayaram Bhasker and Rakesh Chadha. 2009. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer.
[6] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. 2024. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 771–774.
[7] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[8] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[9] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*. 1–7.
[10] Chih-Chun Chang and Tsung-Wei Huang. 2025. Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *ACM/IEEE Design Automation Conference (DAC)*.
[11] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM International Conference on Parallel Processing (ICPP)*. 565–575.
[12] YuAng Chen and Yeh-Ching Chung. 2022. Workload Balancing via Graph Reordering on Multicore Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2022), 1231–1245. doi:10.1109/TPDS.2021.3105323
[13] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 283–284.
[14] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*. 1388–1389.
[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2024. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 766–770.
[16] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*. 468–479.
[17] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–8.
[18] Cheng-Hsiang Chiu, Chedi Morchdi, Chih-Chun Chang, Cunxi Yu, Yi Zhou, and Tsung-Wei Huang. 2025. Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*.
[19] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. 2024. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[20] Yi-Hua Chung, Shui Jiang, Wan Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[21] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[22] Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. 2024. Brief Announcement: PASGAL: Parallel And Scalable Graph Algorithm Library. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 439–441. doi:10.1145/3626183.3660258
[23] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.
[24] Wang Feng, Shiyang Chen, Hang Liu, and Yuede Ji. 2023. PeeK: A Prune-Centric Approach for K Shortest Path Computation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 14 pages. doi:10.1145/3581784.3607110
[25] Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ograss, , and Ali Akoglu. 2025. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*.
[26] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[27] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4219–4232.
[28] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[29] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*. 721–726.
[30] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.
[31] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[32] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 3466–3478.
[33] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[34] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4973–4984.
[35] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.
[36] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–9.
[37] Jin Hu, Greg Schaeffer, and Vibhor Garg. 2015. TAU 2015 Contest on Incremental Timing Analysis: Incremental Timing and CPPR Analysis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design* (Austin, TX, USA) *(ICCAD '15)*. IEEE Press, 882–889.
[38] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–2.
[39] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*. 1–6.
[40] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[41] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing*

Symposium (IPDPS). 746–756.

[42] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[43] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[44] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*. 1–2.

[45] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*. 1360–1363.

[46] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[47] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*. 1–4.

[48] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 757–764.

[49] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 1070–1083.

[50] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. In *IEEE Design and Test (DAT)*.

[51] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.

[52] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[53] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*. 588–597.

[54] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 1–6.

[55] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 758–765.

[56] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*. 51–59.

[57] {Wen mei W.} Hwu, {David B.} Kirk, and {Izzat El} Hajj. 2022. *Programming Massively Parallel Processors: A Hands-on Approach, Fourth Edition*. Elsevier. doi:10.1016/C2020-0-02969-5 Publisher Copyright: © 2023 Elsevier Inc. All rights reserved..

[58] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[59] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM International Conference on Parallel Processing (ICPP)*. 388–399.

[60] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[61] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*. 51–61.

[62] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.

[63] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 278–283.

[64] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–6.

[65] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. 2017. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 273–282.

[66] Pei-Yu Lee, Iris Hui-Ru Jiang, Cheng-Ruei Li, Wei-Lun Chiu, and Yu-Ming Yang. 2015. iTimerC 2.0: Fast incremental timing and CPPR analysis. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 890–894. doi:10.1109/ICCAD.2015.7372665

[67] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, , and Tsung-Wei Huang. 2025. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

[68] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[69] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[70] Wan-Luan Lee, Aditya Das Sarma, Che Chang, Chih-Chun Chang, and Tsung-Wei Huang. 2026. iHyperG: Incremental Hypergraph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

[71] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (2014), 3077–3089.

[72] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*. 2284–2287.

[73] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[74] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 64–71.

[75] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*. 183–188.

[76] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[77] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*. 435–450.

[78] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 3041–3052.

[79] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*. 151–166.

[80] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*. 1–12.

[81] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[82] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[83] Pingchuan Ma, Ziang Yin, Qi Jing, Zhengqi Gao, Nicholas Gangi, Boyang Zhang, Tsung-Wei Huang, Rena Huang, Duane Boning, Yu Yao, and Jiaqi Gu. 2026. SP2RINT: Spatially-Decoupled Physics-Constrained Progressive Inverse Optimization for Diffractive Optical Neural Network Training. In *ACM/IEEE Design Automation Conference (DAC)*.

[84] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. 2009. CUDA Solutions for the SSSP Problem. In *Proceedings of the 9th International Conference on Computational Science: Part I* (Baton Rouge, LA) *(ICCS '09)*. Springer-Verlag, Berlin, Heidelberg, 904–913. doi:10.1007/978-3-642-01970-8_91

[85] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 89–95.

[86] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.

[87] NVIDIA Corporation. 2025. CUDA Core Compute Libraries (CCCL). https://github.com/NVIDIA/cccl. Accessed: 2025-05-27.

[88] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2022. Essentials of Parallel Graph Analytics. In *Proceedings of the Workshop on Graphs,*

Architectures, Programming, and Learning (GrAPL 2022). 314–317. doi:10.1109/IPDPSW55747.2022.00061

[89] David Z Pan, Bill Halpin, and Haoxing Ren. 2008. Timing-driven placement. *Handbook of Algorithms for Physical Design Automation* (2008), 423–446.

[90] Chaitanya Peddawad, Aman Goel, Dheeraj B, and Nitin Chandrachoodan. 2015. iitRACE: A memory efficient engine for fast incremental timing analysis and clock pessimism removal. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 903–909. doi:10.1109/ICCAD.2015.7372667

[91] Eric Ruppert. 2000. Finding the k shortest paths in parallel. *Algorithmica* 28 (2000), 242–254.

[92] Aditya Das Sarma, Shui Jiang, Wan-Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[93] Aditya Das Sarma, Wan-Luan Lee, Shui Jiang, Boyang Zhang, and Tsung-Wei Huang. 2026. A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation. In *ACM/IEEE Design Automation Conference (DAC)*.

[94] Avadhesh Pratap Singh and Dhirendra Pratap Singh. 2015. Implementation of K-shortest Path Algorithm in GPU Using CUDA. *Procedia Computer Science* 48 (2015), 5–13. doi:10.1016/j.procs.2015.04.103 International Conference on Computer, Communication and Convergence (ICCC 2015).

[95] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. 2024. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 789–793.

[96] Jie Tong, Zhengxiong Li, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[97] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*.

[98] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[99] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[100] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[101] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[102] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 190–195.