IncreGPUSTA: GPU-Accelerated Incremental Static Timing Analysis for Iterative Design Flows

Haichuan Liu[1,†], Zizheng Guo[1,2,†], Runsheng Wang[1,2,3], Yibo Lin[1,2,3*]
[1]School of Integrated Circuits, Peking University [2]Institute of EDA, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits
lhaic@outlook.com, {gzz, r.wang, yibolin}@pku.edu.cn

***Abstract*—Static timing analysis (STA) plays an essential role in VLSI design optimization. While CPU-based incremental STA methods reduce computational overhead by selectively updating affected circuit regions, and GPU-accelerated engines improve full-circuit analysis throughput, effectively combining these approaches has remained challenging. Existing solutions offer only partial incrementality, either switching to CPU processing for small modifications or handling solely delay value changes without supporting structural updates. We introduce IncreGPUSTA, a novel GPU-accelerated incremental STA algorithm with dual-CSR data structures and incremental levelization that efficiently processes timing updates for both localized and structural modifications. Experimental results on industrial benchmarks demonstrate speedups of up to 3.06× over GPU full Timer and up to 72.50× over CPU incremental Timer for million-scale designs.**

I. Introduction

Static timing analysis (STA) is a fundamental component of the VLSI design flow, serving as the cornerstone for verifying circuit functionality under given timing constraints [1]. Design optimization algorithms repeatedly invoke STA to iteratively evaluate and improve timing characteristics [2]. Various stages of the design flow, including logic synthesis, placement, routing and physical synthesis, rely heavily on timing feedback after local operations which may impact both the local and global timing landscape [2].

During the optimization process, there are two different timing analysis scenarios: full timing analysis and incremental timing analysis. Full timing analysis involves analyzing the entire circuit after any modification. This approach, while thorough, can be computationally inefficient when changes affect only small portions of the design. In contrast, incremental timing analysis selectively updates only the part of timing data impacted by design changes, as illustrated in Figure 1. Several CPU-based incremental timing engines such as iitRace [3], iTimerC2.0 [4], OpenTimer [5] and OpenTimer v2 [6] have demonstrated considerable runtime improvements through various incremental update strategies. For instance, OpenTimer integrates local updates of affected levels with task-level parallelism to accelerate incremental timing analysis, achieving order-of-magnitude speedup over existing timers [5].

While these CPU-based parallel STA algorithms have made significant advances [7], [5], [6], [8], [9], [10], [11], [12], [13], most are inherently limited by multithreaded parallelism on traditional CPU platforms. Despite their performance benefits, these approaches typically stop scaling beyond 8-16 CPU cores [13], creating a performance ceiling for timing analysis.

[†] Equal contribution, * Corresponding author.

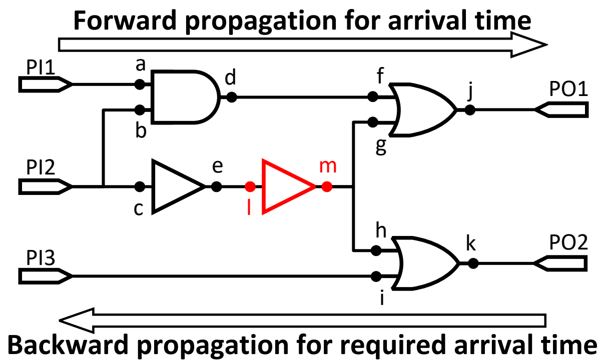

Forward propagation for arrival time
PI1
a
b
d
f
g
j
PO1
PI2
c
e
l
m
h
k
PO2
PI3
i
Backward propagation for required arrival time

Fig. 1: Buffer insertion introduces two new pins m and l. The downstream cone of pin l (l, m, g, h, j, k, $PO1$, $PO2$) requires forward propagation, and the upstream cone of pin m (m, l, e, c, $PI2$) necessitates backward propagation.

To further accelerate timing analysis and overcome limitations of CPU parallelism, researchers have proposed GPU-accelerated STA engines that leverage the massive parallelism of modern GPUs. These approaches have successfully accelerated various time-consuming STA tasks, including delay calculation [14], [15], [16], timing propagation [17], [18], path reports [19], [20], [21], [22], Monte Carlo-based statistical STA (SSTA) algorithms [23], [24], and path exception handling [25]. The performance enhancements achieved by these GPU implementations are substantial when compared to traditional CPU approaches, with documented speedups reaching up to 3.69× for full timing analysis [18] and as high as 25.67× for multi-corner timing analysis [18].

However, GPU-based STA engines encounter significant challenges when applied to incremental timing analysis. Their rigid data structures are designed for bulk operations rather than selective updates, making them ill-suited for iterative design flows. Existing GPU-based approaches that claim "incremental" capabilities are severely limited in their scope of incrementality. For example, Guo et al. [18] merely switch between CPU and GPU processing based on the number of propagation candidates. Similarly, the method presented in [25] only supports updating delay values without accommodating structural modifications. These approaches circumvent the core challenge of incremental updates on GPUs rather than solving it.

To address these limitations, we introduce **IncreGPUSTA**, a novel GPU-accelerated incremental static timing analysis engine. Table I provides a comprehensive comparison between our so-

TABLE I: Comparison of existing static timing analysis engines.

Timer	Incremental	Full	CPU	GPU
OpenTimer [5]	✓	✓	✓	×
OpenTimer v2 [6]	✓	✓	✓	×
OpenSTA [13]	✓	✓	✓	×
GPUTimer [18]	△	✓	✓	✓
HeteroExcept [25]	△	✓	✓	✓
IncreGPUSTA (Ours)	✓	✓	✓	✓

✓: Supported ×: Not supported △: Limited support

lution and existing static timing analysis engines, highlighting IncreGPUSTA's unique position as the first framework to fully support incremental updates (general to both structural and delay modifications) on GPU. Our primary contributions include:

1) A dual-CSR graph representation supporting efficient GPU-based incremental updates, achieving 9.83× speedup over full graph reconstruction approaches.
2) Heterogeneous CPU-GPU update mechanisms that minimize data transfer overhead through batch-processing of modifications into compressed arrays for GPU execution.
3) A three-phase incremental levelization algorithm that reduces computational requirements by 77%, targeting only affected circuit regions through optimized subgraph handling.

We validate IncreGPUSTA in a greedy buffer insertion flow and evaluate its performance using industrial-standard benchmarks from the TAU15 timing contest [2]. Experimental results demonstrate that IncreGPUSTA achieves speedups of up to 3.06× over GPU full Timer [18] and up to 72.50× over CPU incremental Timer [6] for million-scale designs.

The rest of this paper is organized as follows. Section II introduces the background of IncreGPUSTA and a buffer insertion algorithm. Section III details our proposed dual-CSR data structure and incremental update algorithms. Section IV presents our experimental results and analysis. Finally, Section V concludes the paper and discusses future work.

II. Preliminaries

A. Static Timing Analysis

Static Timing Analysis (STA) is a fundamental verification methodology for evaluating signal propagation delays and identifying timing-critical paths within integrated circuits [1]. In the STA process, circuits are modeled as directed acyclic graphs (DAGs) denoted by (V, E), where nodes $v \in V$ represent circuit pins and edges $e \in E$ represent timing arcs associated with logic cells and interconnections [1].

A normal STA engine executes several sequential steps: first, the circuit is levelized to establish a topological ordering, which then enables forward propagation of arrival times (AT) and backward propagation of required arrival times (RAT). This timing information subsequently facilitates further analyses, such as critical path extraction. STA typically employs an early-late split paradigm, where timing data are characterized by both early (minimum) and late (maximum) values [2].

B. Full and Incremental Timing Analysis

Modern digital circuit design involves iterative refinements. Each modification necessitates timing verification, which can be performed by two principal methodologies: full timing analysis and incremental timing analysis.

Full timing analysis recalculates timing information for the entire circuit after each round of modifications. This approach discards all previous computation results. It follows a fixed sequence: circuit graph construction, complete levelization, and exhaustive timing propagation. Despite its thoroughness, this method proves inefficient for iterative workflows, where design changes may affect only small portions of the circuit [2].

Incremental timing analysis employs a more targeted strategy. It preserves previous timing results and selectively updates only data affected by specific modifications. Traditional CPU-based incremental STA engines first identify affected nodes in the timing graph. During forward propagation, only the downstream cones of modified nodes require recalculation. Similarly, backward propagation limits updates to the upstream cones. Timing values for unaffected nodes remain valid, thereby eliminating redundant computation.

C. GPU Incremental STA Challenges

While effective on CPUs, incremental analysis presents unique challenges for GPU implementations. Our performance analysis of GPU Timer [18] on the netcard_iccad circuit yielded an important insight. We discovered that GPU data structure construction consumes approximately 70% of the total runtime, as illustrated in Figure 2. This finding prompted us to develop specialized data structures and update mechanisms that deliver substantial performance improvements.

Notably, we found that for GPU-based implementations, the computational overhead of identifying affected cones may exceed the benefits of selective propagation. This insight suggests that a more promising direction is GPU-accelerated incremental levelization. Conventional STA engines recalculate levelization entirely after each circuit modification. Some tools like OpenTimer [5] implement incremental levelization using bucket list data structures. However, subsequent research [6] identified significant performance bottlenecks in this approach, and shifted towards utilizing task graphs for parallel execution instead.

Moreover, the bucket list structure presents fundamental incompatibilities with GPU architectures. Its dependence on frequent memory allocation and deallocation operations conflicts with GPU memory management principles. Our proposed incremental levelization algorithm directly addresses these challenges. It minimizes device-host transfer overhead while maximizing computational throughput through specialized levelization strategies.

D. Greedy Buffer Insertion

Buffer insertion represents a widely adopted technique in timing optimization for mitigating net delay, thus enhancing setup timing margins (i.e., late slack). While this paper implements a greedy buffer insertion methodology as a case study, the underlying framework is designed to accelerate any iterative, structure-modifying optimization, such as logic synthesis restructuring, physical synthesis optimizations, or the implementation of Engineering Change Orders (ECOs). Our implementation builds upon and extends the buffer insertion algorithms proposed by Van Ginneken [26] and Lillis et al. [27]. To minimize computational overhead while

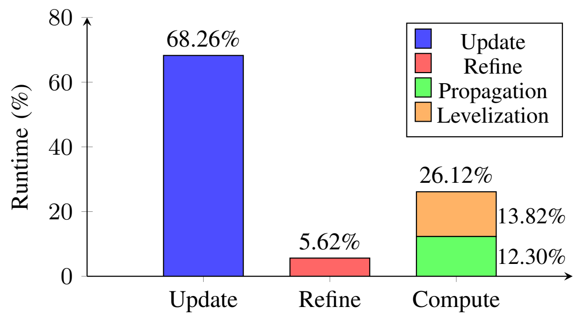


Fig. 2: Runtime breakdown for GPU Timer [18] to complete one round of full timing on a million-gate circuit.

maintaining accuracy, we adopt the Elmore delay model [28], [29] for calculating signal propagation delays.

The Elmore delay model provides a first-order approximation of RC delays in interconnect structures. Its computational efficiency and reasonable accuracy make it a standard choice in numerous STA engines [6], [30], [13]. Within this framework, interconnects are represented as distributed RC networks. The Elmore delay between two pins is mathematically expressed as the product of resistance and downstream capacitive load. This formulation follows the topological methods established in [31].

Our greedy algorithm initially identifies critical nets by analyzing their timing slack values. These slack values are calculated as the difference between arrival time (AT) and required arrival time (RAT). We then select the top k nets for buffer insertion optimization. For each selected net, the algorithm evaluates potential delay reduction at every candidate insertion position. Using the Elmore delay model, we calculate the insertion timing gain for inserting a buffer preceding pin p as follows:

$$Gain = R_{\text{upstream},p} \cdot (C_{\text{load},p} - C_{\text{buf}}) - R_{\text{buf}} \cdot C_{\text{buf}} - delay_{\text{buf}}$$

After completing this analysis across all selected nets, we implement the recorded buffer insertions and update the circuit netlist accordingly. The final step is to recalculate delay values for all affected nets, using GPU-accelerated implementation of the Elmore delay model [18].

III. The IncreGPUSTA Algorithms

In this work, we propose IncreGPUSTA, an efficient incremental algorithm for GPU-accelerated static timing analysis. Building upon previous GPU-accelerated STA frameworks [17], [19], [18], our work focuses on minimizing redundant computation and host-to-device (H2D) data transfers overheads. The core innovations include a dual-CSR graph representation, a heterogeneous update mechanism and a GPU-based incremental levelization algorithm.

As depicted in Fig. 3, the IncreGPUSTA workflow consists of four components: Greedy Buffer Insertion, Incremental CSR Update, Incremental Levelization, and GPU-Accelerated Graph-Net STA. The middle two components—representing our primary technical contributions—are detailed in this section.

A. Dual-CSR Data Structure

The core of our methodology is a dual-CSR graph representation that integrates two complementary CSR matrices with a specialized

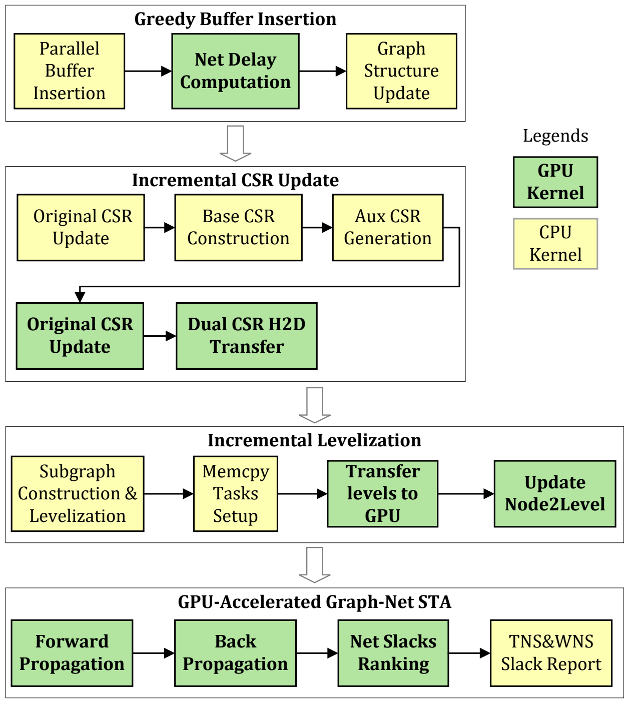


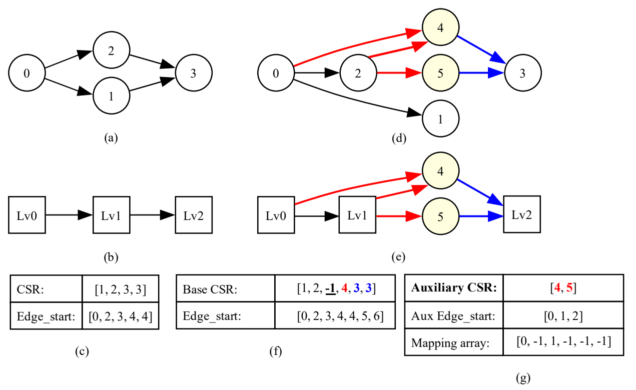
Fig. 3: Our IncreGPUSTA algorithm flow.

mapping array. This architecture decouples stable circuit topology from incremental updates. To illustrate this, we first explain the standard CSR format using the original graph. In the Compressed Sparse Row (CSR) format, a graph's connectivity is stored in two arrays: an offset array and an edge array. For each node i, its adjacency information is stored contiguously in the edge array from the index specified by offset[i] to offset[i+1]-1. For instance, the original DAG in Figure 4(a) is represented by the standard CSR in (c), where the edge array [1, 2, 3, 3] holds all destination nodes and the offset array [0, 2, 3, 4, 4] points to the start of each node's edge list.

When the graph is modified as shown in Figure 4(d), our Dual-CSR structure provides a clear procedure to handle the updates by leveraging a Base CSR and an Auxiliary CSR structure.

The Base CSR matrix is primarily used to manage the foundational graph structure and to incorporate edges emanating from newly introduced nodes. As depicted in Figure 4(f), when nodes 4 and 5 are added, their outgoing edges (4$\rightarrow$3 and 5$\rightarrow$3) are simply appended to the Base CSR's edge array E_B. The offset array O_B is correspondingly extended to map these new nodes to their edges. During initialization, we strategically pre-allocate additional capacity in both arrays to accommodate such future modifications without complete restructuring.

Complementing this foundation, the Auxiliary CSR structure is designed to capture incremental modifications, specifically new edges emanating from existing nodes. This is illustrated in Figure 4(g). To handle the new edges 0$\rightarrow$4 and 2$\rightarrow$5, an Auxiliary Mapping Array (M_A) first maps each original node to its corresponding index in the auxiliary structure (e.g., M_A[0] = 0, M_A[2] = 1) or to -1 if there are no new edges. Then, the auxiliary structure,



CSR:	[1, 2, 3, 3]
Edge_start:	[0, 2, 3, 4, 4]

Base CSR:	[1, 2, **-1**, **4**, **3**, **3**]
Edge_start:	[0, 2, 3, 4, 4, 5, 6]

Auxiliary CSR:	[**4**, **5**]
Aux Edge_start:	[0, 1, 2]
Mapping array:	[0, -1, 1, -1, -1, -1]

Fig. 4: Illustration of the Dual-CSR graph representation and mixed subgraph construction: **(a)** Original DAC. **(b)** Original Level View. **(c)** CSR representation of (a). **(d)** Modified DAC. **(e)** Mixed subgraph View. **(f)** Basic CSR component of (d). **(g)** Auxiliary CSR component of (d).

Algorithm 1: Modification API.

```
1  Function insert_edge(src, dst, delay):
2      if $src$ is an existing node then
3          $inserted\_edges[src].push(\{dst, delay\})$;
4      if $dst$ is an existing node then
5          $inserted\_revedges[dst].push(\{src, delay\})$;
6  Function remove_edge(src, dst):
7      if $dst \in inserted\_edges[src]$ then
8          $inserted\_edges[src].remove(dst)$;
9      else if $src$ is an existing node then
10         $removed\_edges[src].push(dst)$;
11     if $dst$ is an existing node then
12         $removed\_revedges[dst].push(src)$;
```

which replicates the base CSR format with its own offset array O_A and edge array E_A, stores these new connections ([4, 5]) according to the mapping. This approach isolates incremental changes from the stable graph, enabling precise and efficient updates.

B. Data-Centric Heterogeneous Graph Update Strategy

Traditional update approaches regenerate the entire CSR structure on the CPU before transferring the complete data structure to the GPU, incurring substantial performance penalties during iterative design workflows. Our heterogeneous update strategy mitigates this overhead by leveraging the dual-CSR structure for targeted operations on both the CPU and GPU. To support efficient graph traversal on the GPU, this strategy maintains all data structures for both forward edges and reverse edges, denoted by superscripts `out` and `in`, respectively (e.g., E_B^{out} and E_B^{in}).

1) CPU API for Modification: We introduce a modification API that efficiently captures circuit edge changes on the CPU for subsequent CSR updates. Algorithm 1 presents two key functions and four specialized data structures manage these operations: $inserted\_edges$ and $inserted\_revedges$ track new connections, while $removed\_edges$ and $removed\_revedges$ monitor edge deletions. These structures serve as temporary buffers enabling

Algorithm 2: CPU CSR Update Kernel.

```
1  Function update_csr():
2      for $dst, rm\_revedges \in removed\_revedges$ do
3          $ins\_revedges \leftarrow inserted\_revedges[dst]$;
4          for $i \leftarrow 0$ to $len(rm\_revedges) - 1$ do
5              $src \leftarrow rm\_revedges[i]$;
6              $id \leftarrow$ index of $(dst, src)$ in $E_B^{in}$;
7              $E_B^{in}[id] \leftarrow ins\_revedges[i]$;
8              $revedges\_change.push(\{id, ins\_revedges[i]\})$;
9      $n_A \leftarrow 0$;
10     for $src, ins\_edges \in inserted\_edges$ do
11         for $i \leftarrow O_B^{out}[src]$ to $O_B^{out}[src+1]$ do
12             if $E_B^{out}[i] = -1$ then
13                 $E_B^{out}[i] \leftarrow ins\_edges.back()$;
14                 $ins\_edges.pop()$;
15                 $edges\_change.push(\{id, ins\_edges.back()\})$;

16                 if $ins\_edges.empty()$ then
17                     $break$;
18         $rm\_edges \leftarrow removed\_edges[src]$;
19         while $\neg ins\_edges.empty() \wedge \neg rm\_edges.empty()$
            do
20             $dst \leftarrow rm\_edges.back()$;
21             $id \leftarrow$ index of $(src, dst)$ in $E_B^{out}$;
22             $E_B^{out}[id] \leftarrow ins\_edges.back()$;
23             $edges\_change.push(\{id, ins\_edges.back()\})$;
24             $rm\_edges.pop()$;
25             $ins\_edges.pop()$;
26         if $\neg ins\_edges.empty()$ then
27             $nodes\_Aux.push(src)$;
28             $M\_A[src] \leftarrow n_A++$;
29             $O\_A[n_A] \leftarrow O\_A[n_A - 1] + len(ins\_edges)$;
30     for $src, rm\_edges \in removed\_edges$ do
31         for $dst \in rm\_edges$ do
32             $id \leftarrow$ index of $(src, dst)$ in $E_B^{out}$;
33             $E_B^{out}[id] \leftarrow -1$;
34             $edges\_removal.push(id)$;
35     build_auxiliary_csr($nodes\_Aux$, $M_A$);
36     append_base_csr($nodes\_new$);
```

batch processing, facilitating both CPU-side modifications and compact descriptor generation for GPU execution kernels.

2) CSR Update Kernels: The CPU-side update process (Algorithm 2) modifies local CSR structures and generates compact descriptors for GPU kernels. Our strategy is founded on a key property of circuits: the fan-in of a pin remains constant as internal gate connections follow fixed patterns. This stability allows us to employ an efficient substitution strategy for reverse edges: new incoming connections directly replace deleted ones in the base CSR matrix E_B^{in}. This in-place update, shown in lines 2-8 of the algorithm, avoids costly restructuring of the primary graph data.

For forward edges with variable fan-out characteristics, a dynamic approach is required. Our algorithm prioritizes reusing invalidated entries (tracked in *edges_change*, lines 11-25). Any excess insertions are directed to the auxiliary CSR (lines 26-29,

Algorithm 3: CPU CSR Matrices Construction.

```
1  Function append_base_csr($new\_nodes$):
2      for $u \in new\_nodes$ do
3          $O_B^{out}.push(O_B^{out}.back() + len(inserted\_edges[u]))$;
4          $O_B^{in}.push(O_B^{in}.back() +$
            $len(inserted\_revedges[u]))$;
5          for $e \in inserted\_edges[u]$ do
6              $E_B^{out}.push(e)$;
7          for $e \in inserted\_revedges[u]$ do
8              $E_B^{in}.push(e)$;
9  Function build_auxiliary_csr($nodes\_A$, $M_A$):
10     for $u \in nodes\_A$ do
11         for $e \in inserted\_edges[u]$ do
12             $O_A.push(O_A.back() + 1)$;
13             $E_A.push(e)$;
14         for $e \in inserted\_revedges[u]$ do
15             $O_A.push(O_A.back() + 1)$;
16             $E_A.push(e)$;
```

Algorithm 4: GPU CSR Update Kernel.

```
1  Function RemoveEdgesKernel():
2      for $i \leftarrow 0$ to $n\_edges\_removal - 1$ do in parallel
3          $id \leftarrow edges\_removal[i]$;
4          $E_{B,GPU}^{out}[id] \leftarrow -1$;
5  Function ChangeEdgesKernel():
6      for $i \leftarrow 0$ to $n\_edges\_change - 1$ do in parallel
7          $id \leftarrow edges\_change[i].first$;
8          $E_{B,GPU}^{out}[id] \leftarrow edges\_change[i].second$;
9  Function ChangeRevEdgesKernel():
10     for $i \leftarrow 0$ to $n\_revedges\_change - 1$ do in parallel
11         $id \leftarrow revedges\_change[i].first$;
12         $E_B^{in}[id] \leftarrow revedges\_change[i].second$;
13 Function update_gpu():
14     RemoveEdgesKernel();
15     ChangeEdgesKernel();
16     ChangeRevEdgesKernel();
17     copy the appended portion of $E_B^{out}$ to GPU;
18     copy the appended portion of $E_B^{in}$ to GPU;
19     copy $E_A$ to GPU;
```

35), while excess deletions leave invalidated markers (-1) in the base CSR for future use (lines 30-34). New nodes are efficiently incorporated by appending their connectivity to the base CSR (line 36). Finally, the $inserted\_edges$ list is preserved across cycles to facilitate subsequent balancing operations.

To complete our heterogeneous framework, we implement specialized GPU kernels for direct device-side CSR modifications (Algorithm 4). The kernels perform the following tasks: (1) removing edges, (2) changing edges, (3) changing reverse edges, (4) transferring the appended portion of Base CSR to GPU, and (5) transferring the Auxiliary CSR to GPU. Each kernel reads from CPU data structures without writing conflicts, thus eliminating synchronization barriers and ensuring efficient parallelization.

Algorithm 5: Mixed Subgraph Construction

```
1  $n_{mixed\_vertices} \leftarrow n_{new\_vertices} + n_{lv}$;
2  Initialize adjacency lists $edges$ and in-degree array $indeg$;
3  for $i \leftarrow 0$ to $n_{lv} - 2$ do
4      $edges[i].add(i + 1)$;
5      $indeg[i + 1] \leftarrow 1$;
6  for $i \leftarrow 0$ to $n_{new\_vertices} - 1$ do
7      $u \leftarrow i + n_{last\_nodes}$;
8      foreach forward edge $(u, v)$ do
9          if $v < n_{last\_nodes}$ then
10             $l \leftarrow node2level[v]$;
11             $edges[i + n_{lv}].add(l)$;
12             $indeg[l] \leftarrow indeg[l] + 1$;
13         else
14             $edges[i + n_{lv}].add(v - n_{last\_nodes} + n_{lv})$;
15     foreach reverse edge $(v, u)$ do
16         if $v < n_{last\_nodes}$ then
17             $edges[node2level[v]].add(i + n_{lv})$;
18         $indeg[i + n_{lv}] \leftarrow indeg[i + n_{lv}] + 1$;
```

C. Incremental Levelization Algorithm

In GPU-accelerated STA engines, level structures are also encoded in CSR format like edges, with a node array L and an offset array O_L. Typical GPU levelization strategy requires both CPU and GPU to reallocate memory for the arrays and recompute the entire level structure after any design modification.

We address this inefficiency with a three-phase heterogeneous incremental levelization algorithm. This algorithm seamlessly integrates new nodes with existing level data through: (i) mixed subgraph construction, (ii) efficient topological sorting, and (iii) optimized level array reconstruction that minimizes host-device data transfers. This approach maintains topological accuracy while significantly reducing the computational overhead inherent in iterative design workflows. It performs exceptionally well in GPU-accelerated timing analysis, where the number of levels has minimal impact on overall runtime.

1) Mixed Subgraph Construction: The initial phase generates a compact representation that integrates existing level hierarchies with newly introduced nodes (Algorithm 5, Figure 4(e)). Instead of processing the complete circuit graph, we construct a condensed mixed subgraph containing virtual level nodes (indices 0 to $n_{lv} - 1$) that serve as topological proxies for established levels, and actual circuit nodes (indices n_{lv} to $n_{mixed\_vertices} - 1$) representing new components awaiting level assignment.

This condensed representation significantly reduces further computational complexity by scaling with the number of modified elements rather than total circuit size.

2) Mixed Topological Sorting: This phase executes a topological sort on the mixed subgraph to determine optimal level assignments for new nodes(Algorithm 6). Notably, while new nodes appear as $n - n_{last\_nodes} + n_{lv}$ in the edge and in-degree arrays, we maintain the original node identifier n directly in the levels array(lines 12-14). This design choice enables direct host-to-device (H2D) memory transfers while keeping clear distinction between virtual level nodes and actual circuit nodes.

Algorithm 6: Mixed Topological Sorting

```
new_levels ← Initialize empty level vector;
level_0 ← Collect nodes with $outdeg[v] = 0$;
$new\_levels.push\_back(level\_0)$;
for $i \leftarrow 0$ to $new\_levels.size() - 1$ do
    next_level ← Initialize empty node vector;
    for each $u \in new\_levels[i]$ do
        if $u < n\_lv$ then
            $u \leftarrow u - n\_last\_nodes + n\_lv$;
        for each $v \in edges[u]$ do
            $indeg[v] - -$;
            if $indeg[v] = 0$ then
                if $v < n\_lv$ then
                    $v \leftarrow v - n\_last\_nodes + n\_lv$;
                $next\_level.push\_back(v)$;
    if $next\_level.size() > 0$ then
        $new\_levels.push\_back(next\_level)$;
return new_levels;
```

Algorithm 7: Level Array Reconstruction

```
Function NodeToLevelKernel(start, end, level, L, node2level):
    for $i \leftarrow start$ to $end - 1$ do in parallel
        $node2level[L[i]] \leftarrow level$;
Function level_reconstruct(new_levels):
    $new\_n\_lv \leftarrow new\_levels.size()$; $O_L^{new} \leftarrow [0]$;
    $L_{old} \leftarrow$ old level array; $L, h2d\_copies \leftarrow []$
    for $i \leftarrow 0$ to $new\_n\_lv - 1$ do
        $len \leftarrow 0$
        if find a virtual_node in $new\_levels[i]$ then
            $src\_offset \leftarrow O_L[virtual\_node]$;
            $len \leftarrow O_L[virtual\_node + 1] - src\_offset$;
            $D2DCopy(L + O_L^{new}[i], L_{old} + src\_offset, len)$;
            $new\_levels[i]$ remove the $virtual\_node$;
        $O_L^{new}.push(O_L^{new}[i] + len + new\_levels[i].size())$;
        if $new\_levels[i].size() > 0$ then
            $h2d\_copies.push(\{L + O_L^{new}[i] +$
              $len, new\_levels[i], new\_levels[i].size()\})$;
    $sort(h2d\_copies,$ descending by size);
    for $task \in h2d\_copies$ do
        $H2DCopy(task.dst, task.src, task.size)$;
    $O_L \leftarrow O_L^{new}$; $node2level \leftarrow []$;
    for $i \leftarrow 0$ to $new\_n\_lv - 1$ do
        NodeToLevelKernel($O_L^{new}[i]$, $O_L^{new}[i+1]$, i,
          L, node2level);
```

3) Efficient Level Array Reconstruction: The final phase of our algorithm focuses on reconstructing level arrays for both CPU and GPU with efficient copy optimization (Algorithm 7). We integrate virtual nodes into the new level structure and immediately perform device-to-device (D2D) transfers(lines 9-13). Meanwhile, we enhance bandwidth utilization by strategically prioritizing host-to-device (H2D) transfers of new node levels(lines 15-19). This optimization is facilitated by a memory copy task structure that encapsulates specific source/destination pointers and transfer dimensions for efficient batch operations. Finally, we implement a parallel GPU algorithm to reconstruct the node-to-level mapping array(lines 21-22).

TABLE II: Benchmark circuit statistics. For each benchmark, we generated random netlists to achieve critical setup slacks and constructed a random buffer library comprising four buffer types.

Benchmark	# Nodes	# Edges	# Nets	# Pins
leon3mp_iccad	3376832	4039949	649444	5013505
netcard_iccad	3999174	4805566	960615	6979109
leon2_iccad	4328255	5123695	794900	6095880
vga_lcd_iccad	679258	805955	164975	1169323
b19_iccad	782914	1042015	219289	1781026
edit_dist_ispd2	416609	527121	133222	964561
mgc_edit_dist_iccad	450354	560866	133222	964761
matrix_mult_ispd	475186	611390	158526	1164178

IV. Experimental Results

We implemented our GPU-accelerated incremental STA engine **IncreGPUSTA** using GCC 9.3.0 and CUDA 11.5. It was evaluated on an Ubuntu 18.04 Linux server equipped with dual Intel(R) Xeon(R) Gold 6230 CPUs operating at 2.10GHz (20 cores each), 1 NVIDIA GeForce RTX 2080Ti GPU, and 503GB DDR4 RAM. For efficient parallel operations on the GPU and CPU, we leveraged the Thrust library [32] and the TaskFlow library [8], [9], respectively.

We evaluated IncreGPUSTA using a set of benchmarks originated from TAU timing analysis contest [2], with statistics presented in Table II. Our implementation of both full and GPU incremental timer builds upon established GPU-accelerated timing analysis frameworks from prior research [17], [19], [18]. To establish a comprehensive performance baseline, we also use OpenTimer v2 [6] as a reference for CPU incremental timing analysis.

A. Performance and Refinement Comparison

To illustrate the overall performance of IncreGPUSTA, we evaluated two distinct refinement strategies. The first approach targets the top-1000 critical nets in a single pass. The second strategy processes the top-100 critical nets over 10 rounds, better simulating industrial iterative design workflows. Each experiment was repeated three times with results averaged to ensure statistical validity.

Table III demonstrates the significant performance advantages of IncreGPUSTA over alternative implementations. For single-pass refinement scenarios, our approach achieves speedups up to 65.05× compared to the CPU Timer, with an average acceleration ratio of 27.19×. It simultaneously outperforms the GPU full timer by factors up to 2.55×, averaging 1.78× across all benchmarks. The iterative refinement methodology yields even stronger results, with CPU-relative speedups reaching 72.50× (23.95× on average) and GPU-relative improvements up to 3.06× (2.20× on average).

Table IV presents the worst negative slack (WNS) and total negative slack (TNS) for all flip-flops, comparing the original circuits with those refined through greedy buffer insertion, and lists the total number of inserted buffers. Both WNS and TNS show remarkable improvement, indicating that the greedy insertion strategy effectively eliminates setup time violations. The results thus

TABLE III: Total Execution time (ms) and Speed-up Ratio of CPU incremental Timer [6], GPU full Timer [18], and our IncreGPUSTA.

Benchmark	refine top-1000 nets					refine top-100 nets for 10 rounds				
	Total Execution time (ms)			Speed-up Ratio		Total Execution time (ms)			Speed-up Ratio	
	CPU Incre [6]	GPU Full [18]	IncreGPUSTA	$\frac{\text{CPU Incre}}{\text{IncreGPUSTA}}$	$\frac{\text{GPU Full}}{\text{IncreGPUSTA}}$	CPU Incre [6]	GPU Full [18]	IncreGPUSTA	$\frac{\text{CPU Incre}}{\text{IncreGPUSTA}}$	$\frac{\text{GPU Full}}{\text{IncreGPUSTA}}$
leon3mp_iccad_eval	8077.67	331.33	158.67	50.91	2.09	54406.67	2926.67	1082.00	50.28	2.70
netcard_iccad	5786.00	378.67	194.33	29.77	1.95	16110.33	3175.00	1108.00	14.54	2.87
leon2_iccad_eval	9497.00	372.00	146.00	65.05	2.55	76362.33	3223.00	1053.33	72.50	3.06
vga_lcd_iccad_eval	576.33	113.33	71.00	8.12	1.60	6299.33	1048.67	589.00	10.69	1.78
b19_iccad_eval	608.67	151.33	81.33	7.48	1.86	2052.67	1567.33	734.33	2.80	2.13
edit_dist_ispd2	1510.67	108.33	75.33	20.05	1.44	7898.33	801.33	476.67	16.57	1.68
mgc_edit_dist_iccad_eval	1411.33	111.67	77.67	18.17	1.44	8677.00	797.67	500.67	17.33	1.59
matrix_mult_ispd	1482.33	109.33	82.67	17.93	1.32	3148.33	811.67	454.67	6.92	1.79
Avg. Ratio		-		27.19	1.78		-		23.95	2.20

TABLE IV: Setup WNS and TNS (s) of all flip-flops for original and refined designs as well as the total number of inserted buffers.

Benchmark	original		refine top-1000 nets			refine top-100 nets for 10 rounds		
	setup wns (ns)	setup tns (ns)	# bufs	setup wns (ns)	setup tns (ns)	# bufs	setup wns (ns)	setup tns (ns)
leon3mp_iccad_eval	-76.0702	-311960	4455	-28.2327	-107359	17277	-3.77973	-299.070
netcard_iccad	-97.2268	-340247	4019	-57.4648	-138811	11501	-16.6360	-18920.1
leon2_iccad_eval	-59.4002	-163785	1615	-32.3794	-87438.5	4504	-1.70759	-6.77478
vga_lcd_iccad_eval	-89.9565	-209813	5141	-47.7815	-167005	11983	-11.5624	-66370.5
b19_iccad_eval	-53.5467	-99586.8	3147	-42.4668	-94296.0	8980	-22.5999	-80211.8
edit_dist_ispd2	-28.7498	-37304.6	3946	-26.0549	-32962.7	6393	-23.1183	-29639.3
mgc_edit_dist_iccad_eval	-26.4511	-32655.4	4166	-18.4419	-25650.4	6634	-14.8918	-22418.2
matrix_mult_ispd	-25.2580	-25310.8	3590	-22.8098	-19769.5	4481	-18.6961	-18267.0

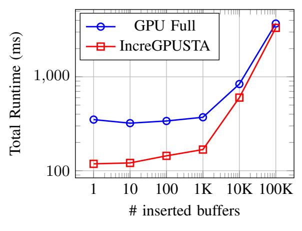
demonstrate that our workloads are from real incremental design update scenarios, not from synthetic test cases.

B. Runtime Scalability

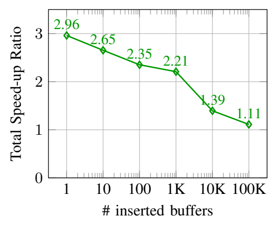
We further analyze the runtime scalability of IncreGPUSTA with respect to the number of inserted buffers, using the netcard_iccad benchmark. Instead of the greedy buffer insertion strategy, we implemented a controlled methodology that randomly selects insertion positions and buffer types. This approach enabled precise control over insertion quantities and timing update scales. We executed each test configuration 3 times and averaged the results to minimize statistical variations. For fair comparison, identical buffer insertion operations were applied to both GPU timers in each experiment.

Figure 5 reveals distinct performance patterns across different buffer insertion scales. When buffer insertion quantities remain significantly below the total net count ($k \ll 960615$), IncreGPUSTA demonstrates superior performance with total speedup ratios between 2.21× and 2.96× for buffer quantities from 1 to 1k. Both timers exhibit relatively stable core runtime because GPU propagation overhead depends primarily on the overall graph size. The actual buffer insertion component remains computationally negligible at these scales.

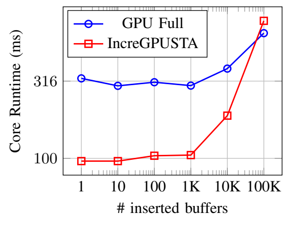
As buffer insertion increases, the performance advantage of the incremental timer gradually diminishes. The total speedup ratio decreases to 1.39× at 10k insertions and approaches parity (1.11×) at 100k insertions. Core runtime analysis reveals two primary factors behind this performance convergence. First, the buffer insertion overhead, which scales linearly with the number of buffers in both implementations, consumes an increasing proportion of the total execution time. Second, while the overheads from kernel launches and memory operations are negligible for smaller workloads, they become progressively more significant at larger scales.



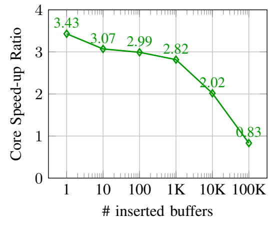
(a) Total Runtime



(b) Total Speed-up Ratio



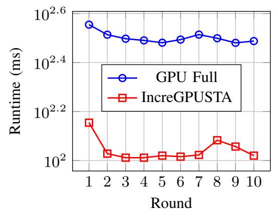
(c) Core Runtime



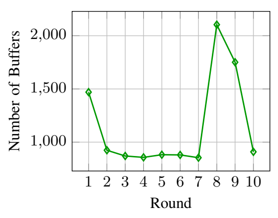
(d) Core Speed-up Ratio

Fig. 5: Performance of GPU full timer vs. IncreGPUSTA on netcard_iccad with increasing inserted buffers: (a) total execution time, (b) total speedup ratio, (c) core runtime (excluding buffer insertion), and (d) core speedup ratio.

These observations identified a threshold effect in GPU-accelerated incremental timing analysis. Below a certain modification-to-total-size ratio, performance is dominated by the fixed overheads of kernel launches and memory management. Beyond these thresholds, incremental timing consistently outperforms full timing approaches—a pattern particularly valuable for timing-driven placement and routing applications [33], [34], [35].

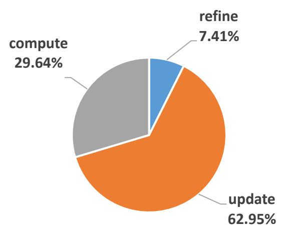


(a) Runtime per Round

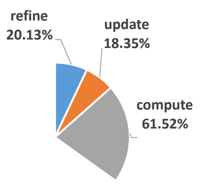


(b) Buffer Insertions per Round

Fig. 6: Detailed runtime and buffer insertion comparison on netcard_iccad using top-100 nets refinement strategy over 10 rounds.



(a) GPU Full (3175 ms)



(b) IncreGPUSTA (1108 ms)

Fig. 7: Runtime breakdown comparing (a) GPU full timer and (b) IncreGPUSTA, demonstrating a 2.87× speedup. Levelization time is incorporated within the compute component.

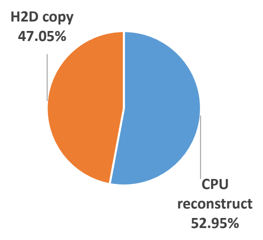
A notable discrepancy emerges between the iterative refinement in Table III and the random insertion experiments in Figure 5. Despite averaging 1,150 buffers per round, our iterative approach achieves a 2.87× speedup—substantially outperforming controlled scenarios with equivalent buffer counts. Detailed analysis of per-round performance in Figure 6 reveals GPU cache warming as the primary contributor to this phenomenon. In IncreGPUSTA, improved cache locality dramatically reduces memory access latencies . The full timer experiences similar cache-related benefits in absolute terms; however, these improvements represent only a negligible fraction of its overall runtime.

C. Runtime Breakdown

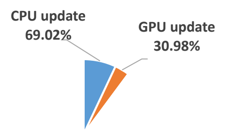
We explore deeper into the performance of IncreGPUSTA by conducting a runtime breakdown analysis compared to the GPU full timer[18]. Using the netcard_iccad benchmark, We focused on the 10-rounds top-100 nets refinement strategy, representing real-world design workflows with frequent local modifications.

Figure 7 illustrates the total runtime profile for both implementations. The GPU full timer requires 3175ms while IncreGPUSTA requires just 1108ms, yielding a 2.87× speedup. Both implementations follow the same workflow: refinement of critical nets, CSR update, and timing propagation. The refinement step consumes minimal time in both cases as it targets only a small part of the design. The most significant difference appears in the CSR update step, which dominates the full timer's execution time (nearly $\frac{2}{3}$ of total) but constitutes less than $\frac{1}{5}$ of the incremental timer's runtime.

Further analysis reveals the source of these performance gains. The timing update phase demonstrates the most dramatic improvement, with IncreGPUSTA achieving a 9.83× speedup as shown in Figure 8. While both implementations involve similar CPU/GPU operation proportions, the full update requires complete CSR structure memory reallocation and transfer—a significant bottleneck eliminated by our incremental algorithm through targeted updates on localized circuit modifications.

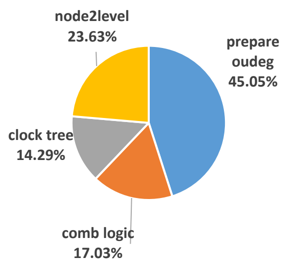


(a) GPU Full (1998.67 ms)

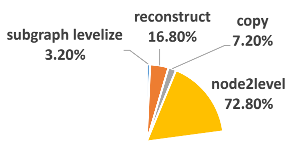


(b) IncreGPUSTA (203.33 ms)

Fig. 8: Update runtime breakdown comparing (a) GPU full timer and (b) IncreGPUSTA, demonstrating a 9.83× speedup.



(a) GPU Full (364 ms)



(b) IncreGPUSTA (83.33 ms)

Fig. 9: Levelization runtime breakdown comparing (a) GPU full timer and (b) IncreGPUSTA, demonstrating a 4.37× speedup.

The levelization process shows additional efficiency gains in runtime, with IncreGPUSTA achieving a 4.37× speedup as illustrated in Figure 9. In the full timer, preparing the out-degree array dominates execution time (nearly $\frac{1}{2}$ of the phase) due to complete circuit traversal. IncreGPUSTA processes only modified components and optimizes memory transfers through strategic use of device-to-device copies for unmodified levels while batching host-to-device transfers for modified levels.

V. Conclusion

This paper presents **IncreGPUSTA**, a GPU-accelerated incremental static timing analysis engine for efficient timing updates in iterative design flows. We employ dual-CSR graph representation, heterogeneous update mechanisms, and an incremental levelization algorithm. Experimental results on industrial benchmarks demonstrate speedups of up to 3.06× over GPU full Timer and up to 72.50× over CPU incremental Timer for million-scale designs.

Acknowledgement

This work was supported in part by the National Science Foundation of China (Grant No. T2293700, T2293701), the Natural Science Foundation of Beijing, China (Grant No. Z230002), and the 111 Project (B18001).

References

[1] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach.* Springer Science & Business Media, 2009.

[2] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2015, pp. 882–889.

[3] C. Peddawad, A. Goel, B. Dheeraj, and N. Chandrachoodan, "iitrace: A memory efficient engine for fast incremental timing analysis and clock pessimism removal," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 903–909.

[4] P.-Y. Lee, I. H.-R. Jiang, C.-R. Li, W.-L. Chiu, and Y.-M. Yang, "iTimerC 2.0: Fast incremental timing and cppr analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2015, pp. 890–894.

[5] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2015, pp. 895–902.

[6] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 40, no. 4, pp. 776–789, 2021.

[7] W. E. Donath and D. J. Hathaway, "Distributed static timing analysis," Apr. 29 2003, uS Patent 6,557,151.

[8] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *International Parallel & Distributed Processing Symposium (IPDPS).* IEEE, 2019, pp. 974–983.

[9] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, vol. 33, no. 6, 2022, pp. 1303–1320.

[10] T.-W. Huang, M. D. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE Design Automation Conference (DAC).* IEEE, 2016, pp. 1–6.

[11] K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *IEEE International Conference on Field-Programmable Technology (FPT).* IEEE, 2018, pp. 110–117.

[12] Z. Guo, T.-W. Huang, and Y. Lin, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," in *ACM/IEEE Design Automation Conference (DAC).* ACM, 2021.

[13] "OpenSTA," https://github.com/The-OpenROAD-Project/OpenSTA.

[14] H. H.-W. Wang, L. Y.-Z. Lin, R. H.-M. Huang, and C. H.-P. Wen, "Casta: Cuda-accelerated static timing analysis for VLSI designs," in *International Conference on Parallel Processing (ICPP).* IEEE, 2014, pp. 192–200.

[15] Z. Guo, T.-W. Huang, Z. Jin, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous static timing analysis with advanced delay calculator," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*, 2024.

[16] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. F. Young, and M. D. Wong, "GCS-Timer: Gpu-accelerated current source model based static timing analysis," in *ACM/IEEE Design Automation Conference (DAC)*, 2024.

[17] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* ACM, 2020.

[18] ——, "Accelerating static timing analysis using cpu-gpu heterogeneous parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2023.

[19] ——, "HeteroCPPR: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* ACM, 2021.

[20] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *ACM/IEEE Design Automation Conference (DAC).* ACM, 2021.

[21] ——, "Gpu-accelerated critical path generation with path constraints," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021, pp. 1–9.

[22] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. F. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2023.

[23] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC).* IEEE, 2009, pp. 260–265.

[24] Y. Shen and J. Hu, "GPU acceleration for PCA-based statistical static timing analysis," in *IEEE International Conference on Computer Design (ICCD).* IEEE, 2015, pp. 674–679.

[25] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, "HeteroExcept: A CPU-GPU heterogeneous algorithm to accelerate exception-aware static timing analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2024.

[26] L. P. Van Ginneken, "Buffer placement in distributed rc-tree networks for minimal elmore delay," in *1990 IEEE International Symposium on Circuits and Systems (ISCAS).* IEEE, 1990, pp. 865–868.

[27] J. Lillis, C.-K. Cheng, and T.-T. Lin, "Optimal wire sizing and buffer insertion for low power and a generalized delay model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 3, pp. 437–447, 1996.

[28] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of applied physics*, vol. 19, no. 1, pp. 55–63, 1948.

[29] R. Gupta, B. Krauter, B. Tutuianu, J. Willis, and L. T. Pileggi, "The elmore delay as bound for rc trees with generalized input signals," in *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, 1995, pp. 364–369.

[30] Y.-M. Yang, Y.-W. Chang, and I. H.-R. Jiang, "iTimerC: Common path pessimism removal using effective reduction methods," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2014, pp. 600–605.

[31] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in rc tree networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 2, no. 3, pp. 202–211, 1983.

[32] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition.* Elsevier, 2012, pp. 359–371.

[33] M.-C. Kim, J. Hu, J. Li, and N. Viswanathan, "Iccad-2015 cad contest in incremental timing-driven placement and benchmark suite," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* IEEE, 2015, pp. 921–926.

[34] N. Viswanathan, G.-J. Nam, J. A. Roy, Z. Li, C. J. Alpert, S. Ramji, and C. Chu, "Itop: Integrating timing optimization within placement," in *Proceedings of the 19th international symposium on Physical design*, 2010, pp. 83–90.

[35] D. Liu, B. Yu, S. Chowdhury, and D. Z. Pan, "Tila-s: Timing-driven incremental layer assignment avoiding slew violations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 231–244, 2017.