HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism

Zizheng Guo
CECA, CS Department
Peking University
Beijing, China
gzz@pku.edu.cn

Tsung-Wei Huang
ECE Department
University of Utah
Salt Lake City, USA
tsung-wei.huang@utah.edu

Yibo Lin*
CECA, CS Department
Peking University
Beijing, China
yibolin@pku.edu.cn

***Abstract*—Common path pessimism removal (CPPR) is a key step to eliminating unwanted pessimism during static timing analysis (STA). Unwanted pessimism will force designers and optimization algorithms to waste a significant yet unnecessary amount of effort on fixing paths that meet the intended timing constraints. However, CPPR is extremely time-consuming and can incur 10–100× runtime overheads to complete. Existing solutions for speeding up CPPR are architecturally constrained by CPU-only parallelism, and their runtimes do not scale beyond 8–16 cores. In this paper, we introduce *HeteroCPPR*, a new algorithm to accelerate CPPR by harnessing the power of heterogeneous CPU-GPU parallelism. We devise an efficient CPU-GPU task decomposition strategy and highly optimized GPU kernels to handle CPPR that scales to large numbers of paths. Also, HeteroCPPR can scale to multiple GPUs. As an example, HeteroCPPR is up to 16× faster than a state-of-the-art CPU-parallel CPPR algorithm for completing the analysis of 10K post-CPPR critical paths in a million-gate design under a machine of 40 CPUs and 4 GPUs.**

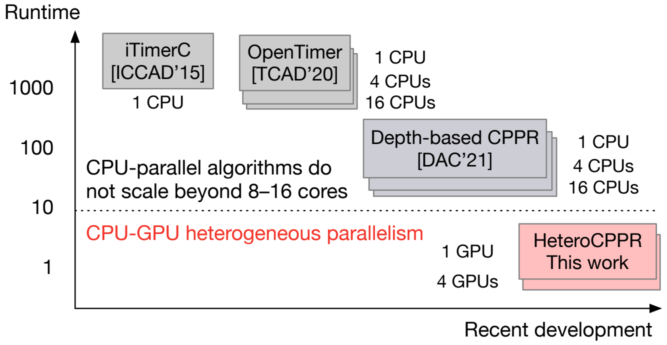

Fig. 1: Recent research on CPPR and its runtime improvement for analyzing a million-gate circuit design using different types of parallelism.

I. Introduction

Static timing analysis (STA) is an important step in the overall VLSI design flow [1]. It analyzes the best-case and worst-case timings of a design and reports timing violations for the given setup and hold tests. This type of analysis enables fast linear-time graph-based analysis (GBA) at a cost of *pessimism* [1]. Specifically, signals do not experience both early and late propagations on the common clock path of the launching path and capturing path for a data path in a flip-flop (FF) pair. *Common path pessimism removal* (CPPR) is thus an imperative step in STA to remove unnecessary pessimism. However, CPPR is extremely time-consuming because it requires enormous path enumerations across all FF pairs. For instance, [2], [3] shows that turning on CPPR can incur 10–100× runtime overheads to complete.

Existing research has been focused on CPU-based algorithms to reduce the long runtimes of CPPR, such as iTimerC [4], iitRace [5], HappyTimer [6], and so on [7], [8]. These algorithms employ different pruning heuristics and parallel decomposition strategies to reduce the search space of arrival time propagation and path generation, and have demonstrated promising speed-up on industrial designs [9]. The popular open-source STA engine, OpenTimer [10], [2], speeds up CPPR by parallelizing independent computations between FF pairs. Guo proposes a new algorithm to replace the enumeration of FFs with clock-tree depth and gains 3–51× speed-up over existing CPPR algorithms [11]. Commercial tools, such as [12], count on tag-based approach to approximate CPPR with linear graph update [9]. While new research continues to speed up CPPR, they are fundamentally limited to CPU parallelism. For large designs of millions of gates and paths, CPPR can still take several hours to complete, resulting significant turnaround time in timing closure. Also, it has been shown that CPU-based CPPR algorithms do not scale beyond 8–16 cores [11].

Figure 1 draws the logarithmic runtimes of state-of-the-art CPPR algorithms and highlights the motivation of this work. To achieve transformational performance milestones for CPPR, new CPPR algorithms must harness the power of *heterogeneous parallelism* comprising manycore CPUs and *GPUs*. Designing a GPU-accelerated CPPR algorithms is not easy, due to the following computational challenges: (1) First, CPPR requires expensive enumerations of FF pairs

*Corresponding author

and/or clock tree levels to identify common segments between launching and capturing paths. This process can peak on giant memory usage that is challenging to fit in relatively limited GPU memory. (2) Second, CPPR is path-specific and graph-oriented. Computations are very irregular and need very specialized data structures and algorithms to exploit GPU parallelism. (3) Third, CPPR needs to analyze a large amount of paths. A practical GPU-accelerated CPPR algorithm must be scalable to multiple GPUs.

In this paper, we propose *HeteroCPPR*, a novel algorithm to overcome the runtime challenges of CPPR by harnessing the power of heterogeneous CPU-GPU computing. HeteroCPPR introduces new GPU-efficient data structures and computation kernels to break the limit of CPU-based scalability. To the best of our knowledge, this is the first work to accelerate CPPR with GPU parallelism. We summarize our technical contributions as follows:

1) We introduce GPU-optimized kernels for critical path generation and pessimism removal. Our kernels address the computational challenges of CPPR and enable fast and accurate CPPR analysis using data parallelism.
2) We introduce GPU-friendly data structures for graph analysis and pruning techniques for path generation. Our algorithms enable efficient GPU memory usage for analyzing large numbers of critical paths.
3) We introduce GPU-scalable parallel decomposition strategies to exploit task parallelism across independent iterations of our CPPR algorithms. Our decomposition strategies scale up the CPPR analysis to multiple GPUs for further performance benefits.

We have evaluated HeteroCPPR on industrial benchmarks from the TAU contests [9]. On a million-gate design, our result using 1 GPU and 1 CPU is 9× faster than the state-of-the-art CPU-based CPPR timer that saturates at 8 CPUs. Using 4 GPUs, we further bring the speed-up up to 19× for analyzing one post-CPPR critical path, and 16× for analyzing 10K post-CPPR critical paths. By leveraging heterogeneous CPU-GPU parallelism, HeteroCPPR can complete the CPPR analysis for tens of thousands of data paths across million-scale circuit benchmarks in just a few seconds.

The rest of this paper is organized as follows. Section II introduces the background of CPPR and its problem formulation. Section III presents details of HeteroCPPR algorithm. Section IV demonstrated the experimental results. Finally, Section V concludes the paper.

II. Preliminaries

A. Common Path Pessimism Removal

STA represents a circuit as a *directed acyclic graph* (DAG), where nodes denote pins and edges denote pin interconnections. Each edge is annotated with a delay value for the signal to pass through. We denote the earliest and the latest arrival time of a pin u as $at^{\text{early}}(u)$ and $at^{\text{late}}(u)$, which are the sums of minimal and maximal delays from a primary input to this pin. A timing path p starts from a primary input and stops at the input pin of a capturing FF (F_c:D), also referred to as the endpoint of a data path p. To ensure that FF F_c captures the input value correctly, we require the following relations between $at(F_c\text{:D})$ and $at(F_c\text{:CK})$ [13]:

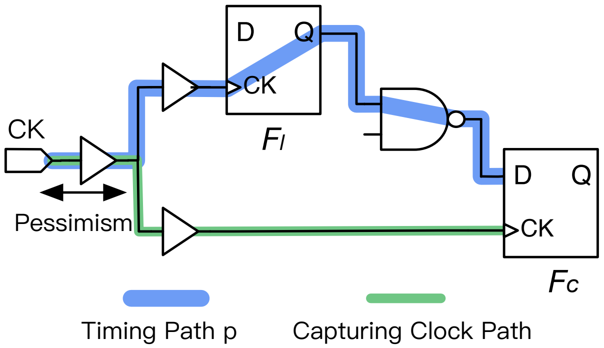

Fig. 2: Example of common path pessimism. The launching clock path and the capturing clock path has a common segment through a buffer that introduces unnecessary pessimism in GBA.

$$\begin{aligned} slack^{\text{setup}}(p) &= at^{\text{early}}(F_c\text{:CK}) + T - T_{\text{setup}} - at^{\text{late}}(F_c\text{:D}), \\ slack^{\text{hold}}(p) &= at^{\text{early}}(F_c\text{:D}) - T_{\text{hold}} - at^{\text{late}}(F_c\text{:CK}), \\ & slack^{\text{setup}}(p), slack^{\text{hold}}(p) \geq 0, \end{aligned} \tag{1}$$

where T, T_{setup}, T_{hold} denote clock period, setup constraint, and hold constraint, respectively.

The two slack values defined in Equation (1) quantify the timing violation of a path p in the worst case. However, as illustrated in Figure 2, the slack values can be overly pessimistic because there exists a common path between the driving path of F_c:D and F_c:CK. Specifically, there is a root clock pin among the primary inputs of the circuit, which drives all FFs in the circuit DAG through a clock tree formed by a subset of pins and edges of the DAG. The common driving path is thus above the lowest common ancestor (LCA) of the launching FF and capturing FF on the clock tree. Edges on this common path accumulate both earliest delay and latest delay in Equation (1), which is not possible and thus introduces pessimism. To fix this unnecessary pessimism, we add a credit to $slack^{\text{setup}}(p)$ and $slack^{\text{hold}}(p)$ as follows [13]:

$$credit(p) = at^{\text{late}}(cp) - at^{\text{early}}(cp), \tag{2}$$

where $cp = LCA(F_l\text{:CK}, F_c\text{:CK})$.

Given the circuit graph and an integer k, the goal of CPPR is to find the top-k paths in an increasing order of their post-CPPR slacks (i.e., decreasing order of criticality).

B. Depth-based CPPR and Pessimism-free Arrival Time

One key challenge of CPPR is that the amount of pessimism is *path-specific*. In other words, we cannot determine the amount of pessimism to remove from a data path without knowing its launching and capturing FFs. Such path-specific pessimism is hard to discover in GBA that cares about only the worst-case timing quantities at each node. To this end, a line of previous works [6], [7], [2], [4], [5], [8] enumerates all pairs

TABLE I: Notations in this paper

Notation	Description
$F_l(p), F_c(p)$	The launching FF and capturing FF of path p.
$f_d(u)$	The first ancestor of node u with depth $\leq d$.
$slack(p)$	The pre-CPPR slack of path p.
$slack(p,d)$	The slack of path p without pessimism above level d.
$AT_d^{\text{setup/hold}}(p)$	The pessimism-free arrival time of path p at level d.

of capturing and launching FFs and performs CPPR analysis for each pair independently. As the number of FFs can be numerous in large designs, these algorithms become difficult to scale and parallelize. Also, storing all pairs of FFs on GPU is not practical due to relatively limited global memory.

In this work, we instead adopt the idea of a recent work by Guo *et al* [11]. They propose to enumerate clock tree depths instead of FFs and demonstrate its efficiencies in practical designs. They identify the common paths by predetermining LCAs and process FF pairs in LCA depth groups. For completeness, we present here the definition of pessimism-free arrival time in [11], using the notations in Table I.

Extended from Equation (1), the slack of path p eliminating the pessimism above clock level d is defined as follows [11]:

$$\begin{aligned} slack(p,d) &= slack(p) + credit(f_d(F_l(p))), \\ &\overset{\text{setup}}{=} at^{\text{early}}(F_c(p)) - AT_d^{\text{setup}}(p) + T - T_{\text{setup}}, \\ &\overset{\text{hold}}{=} AT_d^{\text{hold}}(p) - at^{\text{late}}(F_c(p)) - T_{\text{hold}}, \end{aligned} \tag{3}$$

where the pessimism-free arrival time of path p is defined as,

$$\begin{aligned} AT_d^{\text{setup}}(p) &= at^{\text{late}}(F_l(p)) + delay^{\text{late}}(p) - credit(f_d(F_l(p))), \\ AT_d^{\text{hold}}(p) &= at^{\text{early}}(F_l(p)) + delay^{\text{early}}(p) + credit(f_d(F_l(p))). \end{aligned} \tag{4}$$

In Equation (4), the pessimism-free arrival time is only related to the launching FF, not the capturing FF. As a result, it can be defined for all nodes regardless of the path p considered. The resulting pessimism-free arrival time at timing endpoints can then be used to compute path slacks. The top-k paths ranked by $slack(p,d)$ are called *path candidates*. For different d, different path candidates are computed in independent iterations of the circuit graph. There are a total of $D+2$ iterations, where D denotes the number of clock tree levels [11]. These path candidates can then be combined to yield the top-k paths ranked by post-CPPR slacks (i.e. $slack_{\text{CPPR}}(p)$), which is a central theorem proved in [11].

C. CPPR with Heterogeneous CPU-GPU Parallelism

Although [11] has reported a considerable runtime improvement compared to existing CPPR algorithms, they are all limited to CPU parallelism and do not scale beyond 8–16 cores. To achieve transformational performance milestones, we must incorporate new parallel paradigms comprising *heterogeneous CPUs and GPUs*. However, designing GPU-accelerated CPPR algorithms is challenging. Specifically, we must take into account the distinct performance characteristics and memory architectures of GPU when designing a GPU-accelerated CPPR algorithm. This type of GPU parallelism is very different from CPUs and thus requires very strategic parallel decomposition of the CPPR problem to into data-parallel and memory-efficient GPU tasks. Furthermore, to support large numbers of critical paths, we need to scale the parallel decomposition to multiple GPUs.

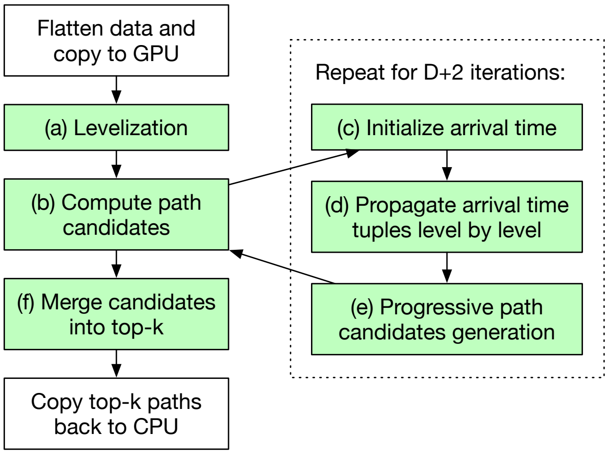
Flatten data and copy to GPU
(a) Levelization
(b) Compute path candidates
(f) Merge candidates into top-k
Copy top-k paths back to CPU
Repeat for D+2 iterations:
(c) Initialize arrival time
(d) Propagate arrival time tuples level by level
(e) Progressive path candidates generation

Fig. 3: Algorithm diagram of HeteroCPPR. Blocks indicate computation tasks, where GPU tasks are colored in green. Arrows indicate task dependencies.

III. Algorithms

In this section, we present our GPU-accelerated CPPR algorithm, HeteroCPPR, to overcome the runtime challenges of CPPR and break the scalability bottleneck of CPU-based parallelism. Figure 3 shows the overview of HeteroCPPR. Our algorithm consists of three steps: (1) *levelization*, (2) *path candidates computation*, and (3) *path candidates merging*. Path candidates computation is done iteratively. In each iteration, there are three substeps: (1) *arrival time initialization*, (2) *propagation*, and (3) *progressive path candidates generation*. All the above steps are implemented on GPU. The circuit data is copied from CPU to GPU at the beginning of computation, and the top-k post-CPPR critical paths are copied back to CPU in the end.

A. Circuit Graph Levelization

The goal of levelization is to build level-by-level dependencies of the circuit graph, including the clock tree network, to facilitate the enumeration of clock tree depths on GPU. The enumerated clock tree depth implies the pessimism for a group of data paths. Specifically, we divide the graph into levels, such that nodes within the same level do not have mutual dependencies, and they only depend on nodes from the previous levels. Based on the levelized graph, we can run the arrival time updates of nodes within the same level in parallel, which forms the basis of GPU-accelerated graph analysis in HeteroCPPR.

Figure 4 shows an example circuit and its levelized layout. A circuit consists of a clock tree and a DAG representing combinational logics. We need to identify the border between the clock tree and the DAG, and levelize each of them separately into a list of levels. We use the levels of the DAG

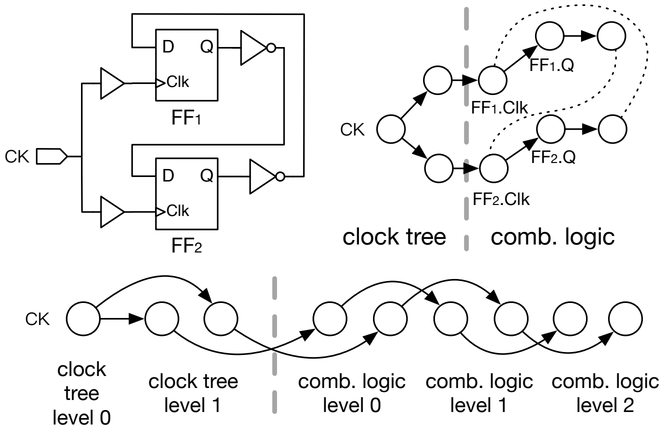

Fig. 4: An example circuit with two FFs, two inverters and two clock buffers. The top-left figure is its logic representation. The top-right figure is its equivalent graph representation. The bottom figure is its levelized layout for GPU computation, which contains 2 clock tree levels and 3 logic levels.

Algorithm 1: Levelization of clock tree and logic nodes

```
1  $D \leftarrow$ array of input degree of nodes;
2  $F \leftarrow$ nodes without output edges;
3  $G \leftarrow \{\}$;
4  while $F$ is not empty do
5      Output a logic level with nodes in $F$;
6      $F' \leftarrow \{\}$;
7      GPU parallel for node $u$ in $F$ do
8          for input edge $v \rightarrow u$ do
9              if atomicAdd($D[v]$, -1) = 1 then
10                 if $v$ is a FF clock pin then add $v$ to $G$;
11                 else add $v$ to $F'$;
12     $F \leftarrow F'$;
13 while $G$ is not empty do
14     Output a clock level with nodes in $G$;
15     $G' \leftarrow \{\}$;
16     GPU parallel for node $u$ in $G$ do
17         There is only one input edge $v \rightarrow u$;
18         Set $u.parent = v$;
19         if atomicAdd($D[v]$, -1) = 1 then
20             Add $v$ to $G'$;
21     $G \leftarrow G'$;
```

Algorithm 2: Group indices propagation

```
1  $D \leftarrow$ the number of clock tree levels;
2  for $i$=0 to $d$ do
3      GPU parallel for node $u$ in clock tree level $i$ do
4          Set $groupid[u]$ as invalid (-1);
5  GPU parallel for node $u$ in clock tree level $d+1$ do
6      $groupid[u] \leftarrow u$;
7  for $i$=$d+2$ to $D-1$ do
8      GPU parallel for node $u$ in clock tree level $i$ do
9          $groupid[u] \leftarrow groupid[u.parent]$;
10 return $groupid$;
```

(i.e. logic levels) to propagate the arrival time of nodes. We use the clock tree levels to update group tags of nodes, based on the fact that each clock tree level is essentially a set of nodes with the same depth.

Algorithm 1 presents our GPU-accelerated levelization algorithm that levelizes both the clock tree and the DAG. Inspired by [14], we first levelize the logic starting from the primary outputs (lines 4-12), by maintaining a set of nodes F indicating the current level of nodes to process and exploring the next F' by a parallel scan of the input edges of F. Different from [14], we need to identify FF clock pins and put them into G instead of F', because they belong to the clock tree. We then levelize the clock tree in a similar fashion as shown in lines 13-21, based on the fact that a node on the clock tree has at most one input edge originating from its parent.

B. Graph-based Analysis: Arrival Time Propagation

Arrival time propagation is the central step in GBA. The goal of arrival time propagation is to compute the worst-case signal arrival time at each node, by the definition in Section II-B. Based on the arrival time of timing endpoints, we can generate critical paths with worst slack values. For a large circuit benchmark, the circuit graph contains millions of nodes, and the propagation is repeated many times to account for different pessimism settings, making it very time-consuming. In this section, we present our GPU-accelerated *arrival time initialization* and *propagation* which correspond to (c) and (d) in Figure 3.

1) Parallel Node Grouping: To ensure that the paths have LCA depth $\leq d$, we use a technique called *node grouping* [11]. Specifically, we group the clock tree nodes by $f_{d+1}(u)$ (see Table I for notations), i.e. based on their ancestor with depth $d{+}1$. Equivalently, we break the clock tree between level d and level $d{+}1$, and regard all subtrees below level $d{+}1$ as groups. An example of node grouping is shown in Figure 5. It has been proved by [11] that two nodes coming from different groups must have LCA depth $\leq d$. In arrival time propagation, we propagate the group indices of launching FFs along with the arrival time to make sure the group constraint can be checked at timing endpoints.

We propose a GPU-accelerated node grouping algorithm that initializes the group indices of all clock tree nodes. We derive two facts about node grouping: (1) every depth-$(d+1)$ node resides in its own group, and (2) every node with depth $> d{+}1$ resides in the same group as its parent. Using these two facts, we propagate the group indices on the clock tree level by level on GPU, as presented in Algorithm 2. First, nodes with depth $\leq d$ are tagged as invalid (lines 2-4) because they do not contribute to the arrival time at level d. Next, depth-$(d{+}1)$ nodes are each assigned its own index as group index (lines 5-6). Finally, the group indices are inherited by the descendants at larger depth (lines 7-9).

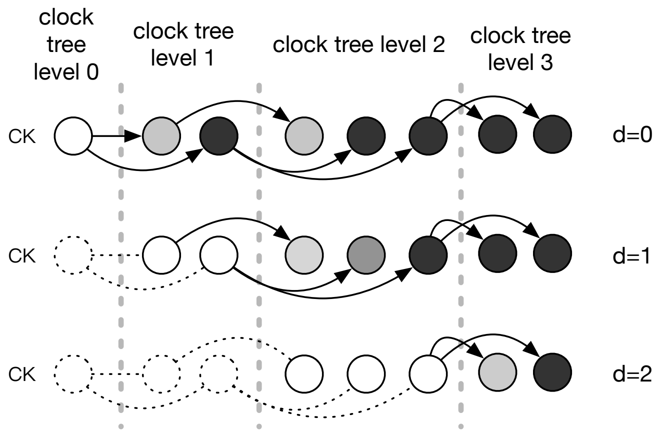

Fig. 5: Illustration of node grouping. The three figures draw the node groups for three LCA depths, $d = 0, 1, 2$, respectively. Different colors indicate different groups.

```
Algorithm 3: Arrival time tuples propagation
1  GPU parallel for node u in DAG do
2      for edge $v \rightarrow u$ where v is a clock tree node do
3          $at \leftarrow at^{late}(v) - credit(f_d(v))$   ▷ for setup;
4          $at \leftarrow at^{early}(v) + credit(f_d(v))$   ▷ for hold;
5          Update $u$'s arrival time tuples with arrival time
             $at$, group index $groupid[v]$, previous node $v$;
6  $D' \leftarrow$ the number of logic levels;
7  for $i$ = 0 to $D' - 1$ do
8      GPU parallel for node $u$ in logic level $i$ do
9          for edge $v \rightarrow u$ where $v$ is a DAG node do
10             for arrival time tuple ($at_0$, $gid_0$, $prev_0$) of
                 $v$ do
11                 $at \leftarrow at_0 + delay^{late}_{v \rightarrow u}$   ▷ for setup;
12                 $at \leftarrow at_0 + delay^{early}_{v \rightarrow u}$   ▷ for hold;
13                 Update $u$'s arrival time tuples with
                     arrival time $at$, group index $gid_0$,
                     previous node $v$;
```

2) Parallel Arrival Time Tuples Propagation: We now describe the detailed arrival time structure and its computation on the DAG through propagation. To allow path recovery, we record the index of the previous node along with the arrival time. To deal with node grouping constraints, we record the group index of the originating launching FF along with the arrival time.

Algorithm 3 shows our GPU-accelerated arrival time tuples propagation algorithm. First, we initialize the arrival time of pins that are directly connected to the clock tree (lines 1-5), according to Equation (4). Then, we propagate the arrival time tuples through successive logic levels (lines 7-13), by enumerating the input edges of all nodes on the DAG, and updating the current arrival time tuples using the arrival time tuples of the predecessor.

C. Path-based Analysis: Parallel Path Generation

The goal of path generation is to produce the top-k paths ranked by post-CPPR slack values, based on arrival times computed in Section III-B. For a complete timing report, the number of paths k to generate can be large, making it the most time-consuming step of CPPR. Path generation on CPU involves highly sequential processes for maintaining path priorities, which is not suitable for GPU execution. As a result, we have to re-design GPU kernels for path generation to benefit from data parallelism. In this section, we propose our fully GPU-accelerated path generation algorithm. The algorithm consists of two steps, which we call (e) *progressive path candidate generation* and (f) *path candidates merging* in Figure 3.

1) Progressive Path Candidates Generation: We represent a path on a timing endpoint by a list of deviation edges from the top-1 path on that endpoint. As shown in Figure 6, the lists of deviation edges are arranged as a *prefix tree*, with the root being the top-1 path, and each node represents a path. This prefix tree representation is widely used in path-based STA [2], [11], [15] as it represents a path using $O(1)$ amount of memory, and it allows us to generate paths progressively.

A typical CPU implementation of the prefix tree algorithm requires maintaining prefix tree nodes using a heap [2], which is inherently sequential and not suitable for GPU parallelization. A recent work on GPU path-based analysis [15] proposes a level-by-level prefix tree expansion algorithm that arranges prefix tree nodes by their levels and uses a set of GPU threads to expand the next level of nodes based on the current level. Inspired by this algorithm, our algorithm for path candidates generation expands an array of prefix tree nodes in parallel, with two major changes to fit with GPU parallelism:

1) Our algorithm generates path candidates instead of paths. We consider group constraints of path candidates in our prefix tree node expansion.
2) Instead of maintaining multiple levels of prefix tree nodes, we only store one array of nodes and update this array iteratively, as shown in Figure 7. This allows us to better prune paths that would not become top-k and complete the algorithm in fewer iterations.

Algorithm 4 shows our progressive path candidates generation algorithm. We start by the top-1 paths at every timing endpoint (lines 3-5). Then, in each iteration, we expand the prefix tree nodes that are recently generated. Before doing the expansion, we allocate memory for nodes to store their expansions, by first asking for their memory demands (lines 10-13) and then doing an in-place prefix sum on the demand array to allocate memory regions (line 14).

After allocating the memory regions, we expand the prefix tree nodes each in a separate GPU thread (lines 15-24). For each prefix tree node, we trace through its shortest path (for hold, longest for setup), and explore the input edges of nodes we visit (lines 18-24). A deviation edge introduces a non-negative cost (line 20), which is the difference between the length of the shortest path and the deviated path. We compute

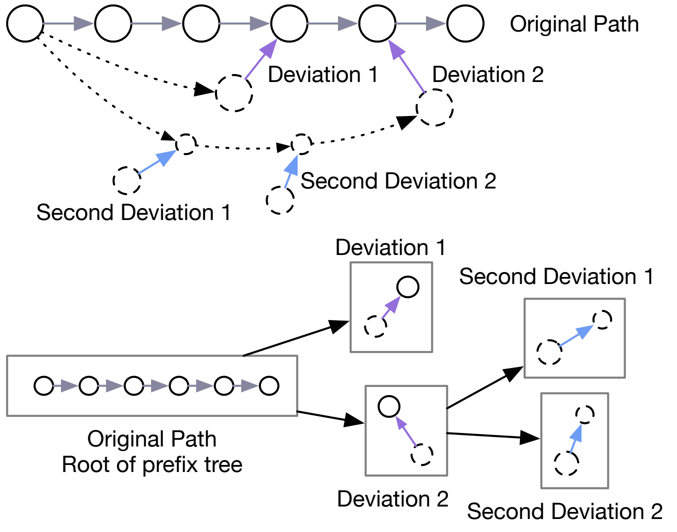

Fig. 6: Illustration of progressive path candidates generation. The top figure shows a path with two deviation edges (tagged Deviation 1 and 2). Each deviation edge introduces a new path, which further introduces other second-level deviation edges (tagged Second Deviation 1 and 2), and so forth. The bottom figure arranges these deviation edges into a prefix tree.

the length of these two paths using the arrival time tuples and the group index of the timing endpoint, which is recorded on the prefix tree node. We compute the slack of new nodes by adding this cost to the slack of old nodes.

We sort the resulting array in increased slack values, using a GPU-based parallel merge sort. After that, we prune the array by keeping the top-k nodes. This is essentially a partial sort where we only care about the order of the smallest k nodes. The prune is done on the first array of top-1 paths (lines 7-8) and on each resulting array after expansion (lines 26-27). Iterations stop until there are no nodes to expand.

2) Path Candidates Merging: In this step, we compute the top-k post-CPPR paths based on all path candidates. They include path candidates from clock tree level d where $0 \leq d \leq D$, self-loop path candidates and primary input path candidates [11]. We compute the first kind of path candidates using Algorithm 3 and 4. We compute the later two kinds of path candidates similarly on GPU. After computing the path candidates, we merge them into a single sorted array of paths and extract the top-k among them. This is a parallel merge of two or more sorted arrays where we only care about the first k elements.

D. Multi-GPU Parallel Iterations

The goal of this step is to scale the proposed GPU algorithm to multiple GPUs for further performance improvement by discovering GPU task parallelism across independent iterations.

In HeteroCPPR, our GPU-accelerated algorithms make use of parallelism within an iteration. To make use of the parallelism between different iterations, we regard GPUs as workers, and the $D + 2$ iterations as independent tasks for

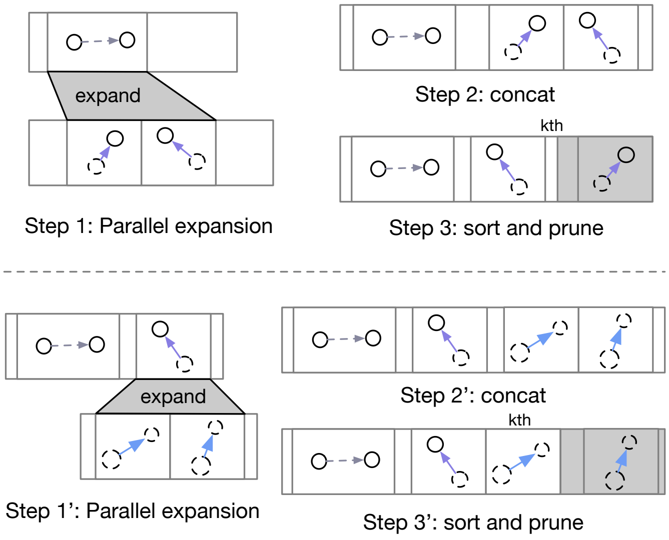

Fig. 7: Illustration of parallel iterative path expansion on GPU using the example in Figure 6. In step 1, we explore the original path for deviation edges and construct two new prefix tree nodes of larger slacks. In steps 2–3, we sort the concatenated array of old nodes and new nodes by their slack values, keep the first k nodes and discard the rest. Notice that the old node is sorted along with the new nodes, pruning more new nodes than just sorting the new nodes.

GPUs to execute in parallel. For example, when 3 GPUs are used, each GPU computes approximately $(D+2)/3$ iterations. Figure 8 shows the overview of our timer using multiple GPUs. Each GPU works on a copy of the circuit graph in its memory. We distribute the circuit graph to multiple GPUs in the beginning, and gather the path candidates in the end to merge them into top-k post-CPPR paths. To reduce the amount of memory transfer between GPUs, we first merge the local path candidates on each GPU. In this way, we need to transfer up to k path candidates from each GPU. Thus, our algorithm can benefit from the computation power of multiple GPUs to further reduce the runtime.

IV. Experimental Results

We implement HeteroCPPR in C++ and CUDA v11.1 and run it on a Linux Ubuntu server with two Intel Xeon CPUs (2.00GHz, 40 cores), 4 GeForce RTX 2080Ti GPUs, and 256 GB RAM. We use the Thrust library [16] to perform parallel sorting on GPUs. We evaluate HeteroCPPR using industrial benchmarks from TAU contests [9]. The benchmark statistics are listed in Table II. We compare HeteroCPPR with the state-of-the-art CPU-based algorithm [11] (“the baseline”) that proposes a provably good and practically efficient methods to speed up CPPR. We do not compare HeteroCPPR with other CPPR algorithms as [11] has demonstrated more than an order of magnitude speed-up over them. Readers can refer to [11] for details about comparisons with other CPU-based CPPR algorithms.

Table III compares the overall runtime performance between the CPU-based CPPR baseline [11] using 8 cores and Het-

Algorithm 4: Progressive path candidates generation

```
$nodes \leftarrow []$;
$slots \leftarrow []$;
GPU parallel for the $i$-th timing endpoint $u$ do
    compute $slack$ according to Equation (3);
    $nodes[i] \leftarrow$ new prefix tree root node with level=1,
      slack=$slack$, where=$u$, gid=$groupid[u]$;
$curlevel \leftarrow 1$;
GPU parallel sort $nodes$;
$nodes \leftarrow nodes[0:k]$;
repeat
    GPU parallel for the $i$-th node $p$ in $nodes$ do
        if $p.level == curlevel$ then
            $slots[i] \leftarrow$ estimated number of new nodes
              to expand from $p$;
        else $slots[i] \leftarrow 0$;
    GPU parallel in-place prefix sum on $slots$;
    GPU parallel for the $i$-th node $p$ in $nodes$ with
      $p.level == curlevel$ do
        $u \leftarrow p.where$;
        $j \leftarrow k + slots[i]$;
        repeat
            for deviation edge $v' \rightarrow u$ do
                $cost \leftarrow at(v', p.gid) + delay_{v' \rightarrow u} -$
                  $at(v, p.gid)$;
                $nodes[j] \leftarrow$ new prefix tree node with
                  parent=$p$, level=$curlevel + 1$,
                  slack=$p.slack + cost$, where=$u$,
                  gid=$p.gid$;
                $j \leftarrow j + 1$;
            $u \leftarrow$ the previous node in $u$'s arrival time
              tuples with different group index than
              $p.gid$;
        until $u$ is a clock tree node;
    $curlevel \leftarrow curlevel + 1$;
    GPU parallel sort $nodes$;
    $nodes \leftarrow nodes[0:k]$;
until no new node is expanded;
```

eroCPPR using 1, 2, 3, and 4 GPUs. We use 8 cores for the baseline where its performance saturates. Compared to the baseline, HeteroCPPR with 1 GPU achieves an average speed-up of 8.15×, 6.95×, 6.23×, and 5.11× for generating 1, 100, 1K, and 10K post-CPPR critical paths, respectively. The speed-up becomes even more remarkable when using multiple GPUs. For example, HeteroCPPR with 2 GPUs achieves 12.57×, 10.93×, 9.93×, 8.28× average runtime speed-ups over the baseline for generating 1, 100, 1K, and 10K post-CPPR critical paths, respectively. With 4 GPUs, HeteroCPPR brings up the speed-ups to 14.61×, 13.17×, 12.11×, and 10.52×. For large benchmarks, the speed-up over baseline becomes even remarkable. HeteroCPPR is up to 9.95× faster (`leon3mp`, $k = 1$) using 1 GPU and 19.46× (`netcard`, $k = 1$) using 4 GPUs. For `netcard` with $k = 10K$, HeteroCPPR is up to 7.51× and 16.61× faster under 1 GPU and 4 GPUs, respectively. These runtime results highlight the performance advantage of HeteroCPPR.

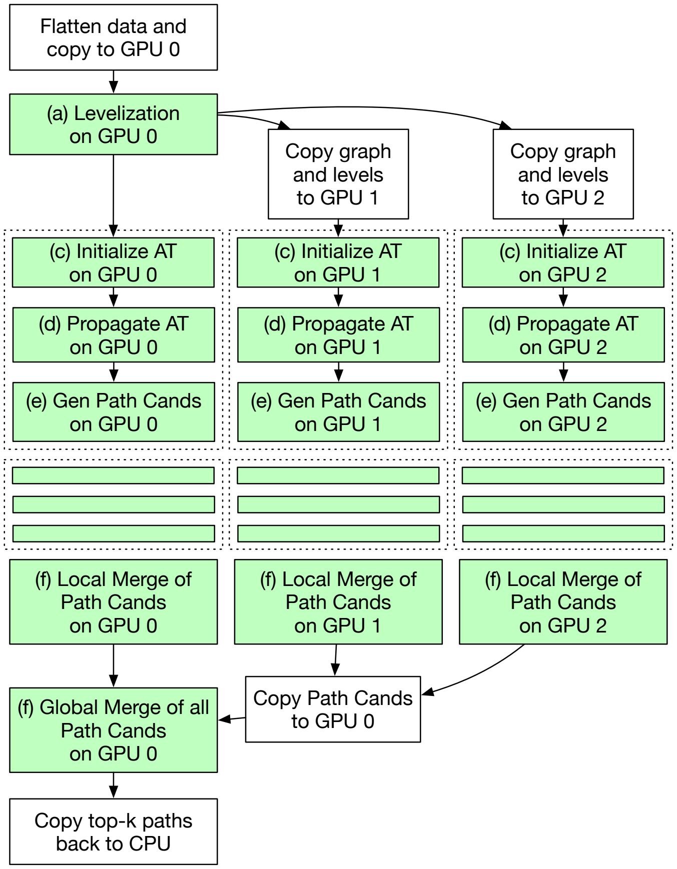
Flatten data and copy to GPU 0
(a) Levelization on GPU 0
Copy graph and levels to GPU 1
Copy graph and levels to GPU 2
(c) Initialize AT on GPU 0
(c) Initialize AT on GPU 1
(c) Initialize AT on GPU 2
(d) Propagate AT on GPU 0
(d) Propagate AT on GPU 1
(d) Propagate AT on GPU 2
(e) Gen Path Cands on GPU 0
(e) Gen Path Cands on GPU 1
(e) Gen Path Cands on GPU 2
(f) Local Merge of Path Cands on GPU 0
(f) Local Merge of Path Cands on GPU 1
(f) Local Merge of Path Cands on GPU 2
Copy Path Cands to GPU 0
(f) Global Merge of all Path Cands on GPU 0
Copy top-k paths back to CPU

Fig. 8: The overall taskflow of HeteroCPPR using multiple GPUs. This figure shows an example using 3 GPUs, demonstrating the dependency of computation and data transfer operations.

TABLE II: Benchmark statistics. This same set of benchmarks were used in [11] to compare different CPPR algorithms on CPU.

Benchmark	# Edges	# FFs	# Levels (D)
vga_lcdv2	449651	25091	56
Combo4v2	778638	26760	82
Combo5v2	2051804	39525	91
Combo6v2	3577926	64133	101
Combo7v2	2817561	54784	96
netcard_iccad	3999174	97831	75
leon2_iccad	4328255	149381	85
leon3mp_iccad	3376832	108839	75

Figure 9 compares the runtimes between the CPU baseline and HeteroCPPR for generating 1K post-CPPR critical paths using different numbers of CPUs and GPUs. We can clearly see that the performance of the CPU baseline does not scale beyond 8 cores. Besides, we observe a consistent performance gap between the baseline and HeteroCPPR, regardless of the numbers of CPU cores and GPUs. For example, in `netcard`, our algorithm with 1 GPU is 6.10× faster than the baseline

TABLE III: Overall runtime comparisons between the state-of-the-art CPU-based CPPR timer [11] and HeteroCPPR under different k (post-CPPR critical paths) and numbers of GPUs. Runtime is measured in milliseconds.

Benchmark	k=1									k=100								
	CPU [11]	GPU				Speed-up Ratio				CPU [11]	GPU				Speed-up Ratio			
	8 cores	×1	×2	×3	×4	×1	×2	×3	×4	8 cores	×1	×2	×3	×4	×1	×2	×3	×4
vga_lcdv2	836	168	110	98	98	4.98	7.60	8.53	8.53	853	230	152	130	124	3.71	5.61	6.56	6.88
Combo4v2	1624	315	214	181	172	5.16	7.59	8.97	9.44	1594	436	289	238	224	3.66	5.52	6.70	7.12
Combo5v2	4107	504	348	313	315	8.15	11.80	13.12	13.04	4078	632	420	371	354	6.45	9.71	10.99	11.52
Combo6v2	7244	765	516	462	484	9.47	14.04	15.68	14.97	7482	914	603	538	539	8.19	12.41	13.91	13.88
Combo7v2	5546	616	427	390	392	9.00	12.99	14.22	14.15	5795	772	501	457	452	7.51	11.57	12.68	12.82
netcard	12627	1347	820	681	649	9.37	15.40	18.54	19.46	12265	1430	853	706	673	8.58	14.38	17.37	18.22
leon2	13327	1456	870	726	703	9.15	15.32	18.36	18.96	12984	1548	936	766	735	8.39	13.87	16.95	17.67
leon3mp	9491	954	599	523	517	9.95	15.84	18.15	18.36	9445	1036	656	565	547	9.12	14.40	16.72	17.27
Avg. Ratio	-					8.15	12.57	14.45	14.61	-					6.95	10.93	12.73	13.17

Benchmark	k=1,000									k=10,000								
	CPU [11]	GPU				Speed-up Ratio				CPU [11]	GPU				Speed-up Ratio			
	8 cores	×1	×2	×3	×4	×1	×2	×3	×4	8 cores	×1	×2	×3	×4	×1	×2	×3	×4
vga_lcdv2	875	286	192	164	155	3.06	4.56	5.34	5.65	1027	409	281	239	228	2.51	3.65	4.30	4.50
Combo4v2	1616	542	344	293	270	2.98	4.70	5.52	5.99	1842	880	581	537	536	2.09	3.17	3.43	3.44
Combo5v2	4454	761	500	415	394	5.85	8.91	10.73	11.30	4372	1058	671	552	513	4.13	6.52	7.92	8.52
Combo6v2	7271	1011	654	579	591	7.19	11.12	12.56	12.30	7900	1244	791	666	653	6.35	9.99	11.86	12.10
Combo7v2	5640	911	589	515	506	6.19	9.58	10.95	11.15	6201	1210	771	648	620	5.12	8.04	9.57	10.00
netcard	12592	1499	897	735	703	8.40	14.04	17.13	17.91	13168	1753	1037	838	793	7.51	12.70	15.71	16.61
leon2	13219	1670	990	816	768	7.92	13.35	16.20	17.21	13513	2028	1196	966	895	6.66	11.30	13.99	15.10
leon3mp	9551	1157	723	601	587	8.25	13.21	15.89	16.27	10030	1536	923	768	724	6.53	10.87	13.06	13.85
Avg. Ratio	-					6.23	9.93	11.79	12.22	-					5.11	8.28	9.98	10.52

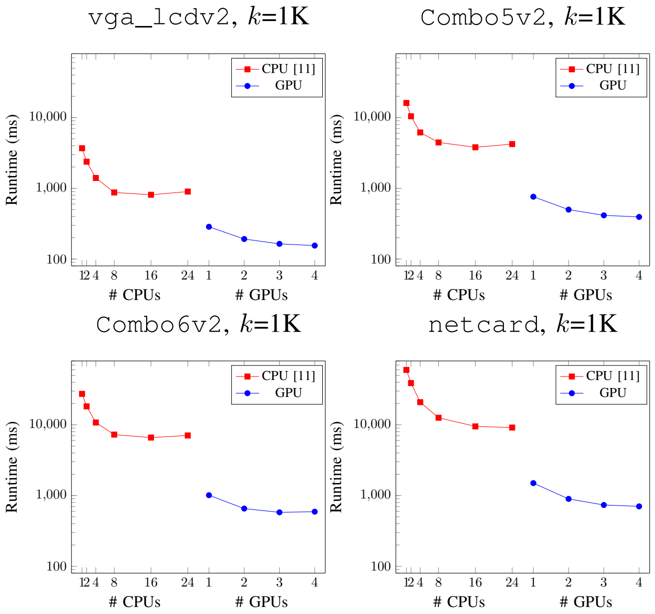

Fig. 9: Runtimes at different numbers of CPU cores and GPUs.

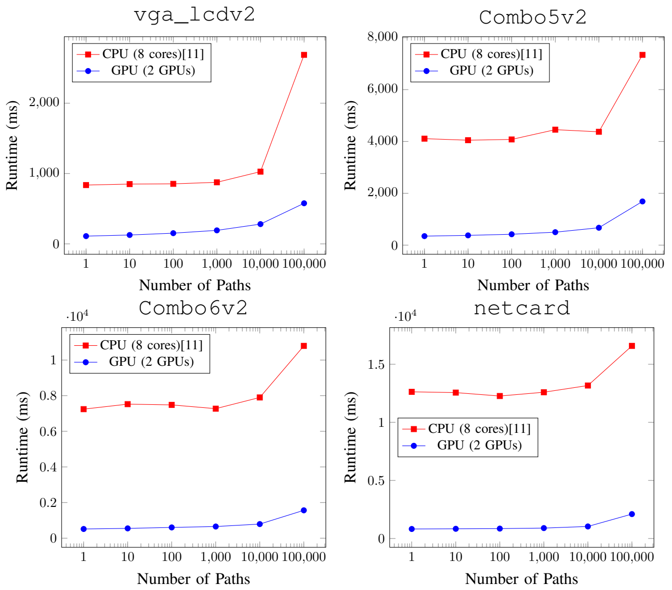

Fig. 10: Runtimes at different numbers of post-CPPR paths (i.e., k).

with 24 cores. The results show that HeteroCPPR is able to reduce the long runtime of CPPR with new performance milestones by harnessing the power of GPU computing.

Figure 10 compares the runtimes between the CPU baseline and HeteroCPPR for generating different numbers of post-CPPR paths. We assign 8 cores to the CPU baseline and 2 GPUs to HeteroCPPR, in which both perform close to the best runtimes. Similar to Figure 9, we observe a consistent performance gap between the baseline and HeteroCPPR, regardless of the numbers of generated post-CPPR paths. Also, the gap continues to enlarge as we increase the number of paths. For instance, in `vga_lcdv2`, when increasing the number of generated post-CPPR paths from 10K to 100K, the runtime growth of HeteroCPPR is only 2.05× (281 vs 577), whereas the baseline is 2.61× (1027 vs 2687). Similar trends can be observed in other three benchmarks, `Combo5v2`, `Combo6v2`, and `netcard`. This scalability result highlights the effectiveness of our GPU kernels and iterative path generation.

V. Conclusion

In this paper, we have introduced *HeteroCPPR*, a novel algorithm to accelerate CPPR by harnessing the power of heterogeneous CPU-GPU parallelism. We have devised an efficient CPU-GPU task decomposition strategy and highly optimized GPU kernels to speed up critical computational tasks of CPPR that scales to large numbers of paths. Our parallel decomposition strategies can scale to multiple GPUs to further bring up the speed-up to another degree. As an example, HeteroCPPR is up to 16× faster than a state-of-the-art CPU-parallel CPPR algorithm for completing the analysis of 10K post-CPPR critical paths in a million-gate design under a machine of 40 CPUs and 4 GPUs. We have also shown that for large designs, regardless of the number of CPUs or the number of generated post-CPPR ciritical paths, HeteroCPPR is consistently faster than the CPU baseline and the performance gap becomes even remarkable at large problem sizes. Our future work plans to investigate new GPU task graph parallelism, CUDA graph, to further improve the performance of HeteroCPPR.

ACKNOWLEDGE

This work was supported in part by the National Science Foundation of China (Grant No. 62004006 and No. 62034007).

REFERENCES

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

[2] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, pp. 1–1, 2020.

[3] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *Proc. IPDPS*. IEEE, 2019, pp. 974–983.

[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 *Proc. ICCAD*. IEEE, 2015, pp. 890–894.

[5] C. Peddawad, A. Goel, B. Dheeraj, and N. Chandrachoodan, "iitrace: A memory efficient engine for fast incremental timing analysis and clock pessimism removal," in *Proc. ICCAD*, 2015, pp. 903–909.

[6] B. Jin, G. Luo, and W. Zhang, "A fast and accurate approach for common path pessimism removal in static timing analysis," in *Proc. ISCAS*. IEEE, 2016, pp. 2623–2626.

[7] T.-W. Huang and M. D. Wong, "UI-timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.

[8] T. Chung-Hao and M. Wai-Kei, "A fast parallel approach for common path pessimism removal," in *Proc. ASPDAC*, 2015, pp. 372–377.

[9] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.

[10] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.

[11] Z. Guo, T.-W. Huang, and Y. Lin, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," in *Proc. DAC*. ACM, 2021.

[12] "OpenSTA," https://github.com/abk-openroad/OpenSTA.

[13] J. Hu, D. Sinha, and I. Keller, "TAU 2014 contest on removing common path pessimism during timing analysis," in *Proc. ISPD*, 2014, pp. 153–160.

[14] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.

[15] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *Proc. DAC*. ACM, 2021.

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