Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism

Zizheng Guo, Tsung-Wei Huang *Member, IEEE*, Yibo Lin *Member, IEEE*

***Abstract*—Static timing analysis (STA) is an essential yet time-consuming task during the circuit design flow to ensure the correctness and performance of the design. Thanks to the advancement of general-purpose computing on graphics processing units (GPUs), new possibilities and challenges have arisen for boosting the performance of STA. In this work, we present an efficient and holistic GPU-accelerated STA engine. We accelerate major STA tasks, including levelization, delay computation, graph propagation, and multi-corner analysis, by developing high-performance GPU kernels and data structures. By dividing the STA workloads into CPU-GPU concurrent tasks with managed dependencies, our acceleration framework supports versatile incremental updates. Furthermore, we have extended our approach to multi-corner analysis by exploring a large amount of corner-level data parallelism using GPU computing. Our implementation based on the open-source STA engine OpenTimer has achieved up to 4.07× speed-up on single corner analysis, and up to 25.67× speed-up on multi-corner analysis on TAU 2015 contest designs and a 14nm technology.**

I. Introduction

With the advancement of design complexities and process technology, the overall circuit design closure is increasingly bounded by the timing analysis on circuit graphs consisting of hundreds of process corners and billions of transistors. To ensure the timing correctness and performance of the design, *static timing analysis* (STA) is frequently invoked in the iterative and incremental updates inside optimization algorithms [1]. In response to millions of design modifications performed by the optimization flow, the timer is required to provide instant and accurate feedback on slack values and timing criticality changes. To achieve acceptable performance and design turnaround time, it is crucial to have an efficient STA engine.

A number of parallel STA algorithms have been proposed in previous works, including both commercial tools and academic research [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. While each has their advantages and drawbacks, nearly all of these algorithms are inherently bound by the *multithreaded* parallelism on a platform with central processing units (CPUs) and multi-core architecture. While some of these attempts have gained runtime benefits, most of them stop scaling beyond 8–16 CPU cores [11], without more scalable replacements above that. With the increasing computing capacity of modern graphics processing units (GPUs), new possibilities have arisen for boosting the performance of STA engines. However, it is extremely challenging to develop an efficient STA engine running on a CPU-GPU heterogeneous platform. Computing the signal timing across a circuit graph involves both diverse computational patterns and irregular memory access, including dynamic data structures, graph-oriented computing, recursion, and branch-and-bound, to name a few [3]. These patterns lead to a vast and complex STA workload consisting of nontrivial functional dependencies. We need very strategic data structure models and decomposition algorithms to obtain a reasonable STA runtime speed-up.

As design shifts to nanoscale, the timing effects from process, voltage, and temperature (PVT) changes are more and more tied to the successful tapeout of chips. The analysis of such effects is done by *multi-corner* STA engines that address all combinations of PVT corners in independent STA runs, which bear a large memory footprint and huge runtime. Most previous research on multi-corner STA acceleration [12], [13], [14], [15], [16], [17] is limited to CPU-based parallelism either within one or multiple STA processes. This organization cannot efficiently explore large data parallelism exhibited by multi-corner settings.

In this work, we present a new STA implementation on a CPU-GPU heterogeneous platform. We propose GPU-efficient acceleration kernels and data structures to offload major STA computing steps to GPU. We implement our algorithms based on the open-source STA engine OpenTimer, designed by Huang *et al* [3]. Our algorithm's core design philosophy is universally applicable and can be applied to other STA frameworks. The major contributions of this paper are summarized in the following:

- We divide the STA workloads into CPU-GPU concurrent tasks with managed dependencies by leveraging task-based parallelism, effectively hiding data processing and memory latency.
- We develop high-performance GPU kernels and data structures tailored to GPU parallelism for all major STA operations including delay calculation, levelization, and graph propagation.
- We implement our GPU-accelerated STA algorithms based on a real-world STA framework with support to industrial design formats and incremental timing. Our techniques provide valuable insight into CPU-GPU per-

The preliminary version, entitled "GPU-Accelerated Static Timing Analysis", has been presented at the International Conference on Computer-Aided Design (ICCAD) in 2020. This work was supported in part by the National Science Foundation of China (Grant No. 62034007 and No. 62004006), National Science Foundation of US (CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, and TI-2229304.), and the 111 Project (B18001).

Z. Guo is with the School of Integrated Circuits, Peking University, Beijing, China. Y. Lin is with the School of Integrated Circuits, Peking University, Beijing, China, Institute of Electronic Design Automation, Peking University, Wuxi, China, and Beijing Advanced Innovation Center for Integrated Circuits, Beijing, China. Corresponding author: Yibo Lin (yibolin@pku.edu.cn).

T.-W. Huang is with the Department of Electrical and Computer Engineering, the University of Wisconsin at Madison, Wisconsin, USA.

formance tradeoff in realistic scenarios.

- We extend our GPU algorithms to multi-corner timing analysis by exploring data parallelism across the corner dimension and proposing efficient memory mapping under GPU memory constraints. We have demonstrated a substantial extra performance benefit.

We evaluate our algorithm on the gate-level circuit netlists from TAU 2015 Timing Analysis Contest benchmarks consisting of large industrial designs [18]. We use a 14 nm technology to provide realistic single-corner and multi-corner cell libraries. We have achieved a significant speed-up on both single-corner and multi-corner STA. As an example, on two large circuit designs, netcard and leon2, we accelerate OpenTimer by 4.05× and 4.07× using one GPU for single-corner analysis. By computing 16 corners in parallel, we achieved another 5.66× and 6.34× speed-up on these designs. We also investigated the impact of several factors on incremental timing performance, such as gate number, net count, and incremental graph size, and provided recommendations on when to use GPU or CPU. Given the composite of software tradeoffs and architectural considerations we have made, we believe our STA algorithm delivers a novel acceleration methodology.

The organization of this paper is listed as follows. In Section II , we introduce the STA background, GPU architecture, and our problem formulation. In Section III, we present details of our multi-corner STA algorithm on GPU. In Section IV, we demonstrate the experimental results on STA runtime improvement. Section V summarizes the paper.

II. Static Timing Analysis

In STA, circuits are represented as *directed acyclic graphs* (DAGs), where nodes represent pins of circuit components and edges represent pin interconnects. Figure 1 illustrates an example STA graph consisting of 4 logic cells, 5 primary ports, and 7 nets. In order to compute the signal arrival times on this timing graph, a *graph-based* STA engine performs two major steps called *forward* and *backward propagation*. In forward propagation, timing quantities such as delay, slew, and RC are computed, and arrival time (AT) is accumulated according to data dependencies. In backward propagation, timing constraints and slack statistics such as required arrival time (RAT) are computed based on the forward propagation result. To keep our discussion focused, we compute cell delays through the nonlinear delay model (NLDM) based on 2D look-up tables (LUTs) with load and slew indices, and net delays through the widely known Elmore delay model. These models are used in recent TAU 2014–2019 timing analysis contests [19], [18], [20] with golden results available. The node criticality is quantized by its *slack* defined as the difference between its AT and RAT. Setup check and hold check, respectively, refer to the late and early slack values.

A. Parallel STA Engines

The design complexity of modern VLSI systems are ever-increasing, with millions to even billions of pins and interconnect. Such a large amount of computation puts unprecedented pressure on the analysis speed of STA engines

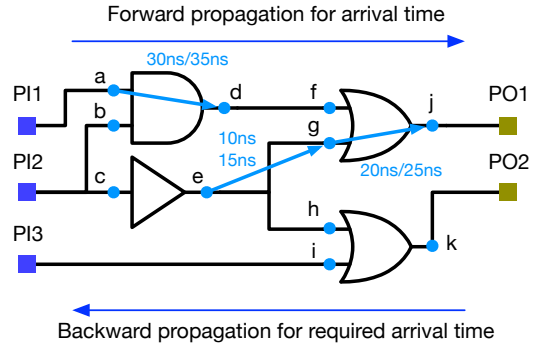


Fig. 1: The STA graph of a circuit design. The blue nodes indicate pins of components like gates and I/O ports. The arrows indicate pin-to-pin connections. Delay values are quantified using best (min) and worst (max) scenarios.

used to evaluate large designs. To shorten STA's substantial runtime, ongoing research has looked towards parallel and distributed STA frameworks [2], [3], [6], [7]. Huang *et al*, for example, developed OpenTimer, a timing analysis engine that represents timing propagation jobs and their precedence using a dependency graph of tasks [3], [6], [5]. Their result has demonstrated up to 2× runtime improvement over loop-based parallelization using OpenMP. Another attempt on parallel timing graph propagation for FPGA designs is performed by Murray *et al* and demonstrated 9× speed-up using 32 CPU cores. Besides, new challenges on timing macro-modeling [21], [22], common path pessimism removal [8], [9], [10], and incremental timing are also raised by recent TAU contests [19], [18], [20].

The multi-threading performance based on CPUs generally saturates at roughly 8–16 threads, due to irregular computational patterns and threading overhead of STA [3], [7]. To overcome the scalability challenge, timing analysis with GPU acceleration is further investigated [23], [7]. Wang *et al* [23] have explored GPU acceleration of the LUT interpolation step during cell delay computation while leaving other STA steps like levelization and net delay computation on CPU. Regarding the kernel runtime, a more than 10× speed-up has been demonstrated compared to their CPU version. The work by Murray *et al* [7] mentioned before also investigated timing propagation on GPUs. While their kernel runtime has been 6.2× faster, the overall propagation runtime has become even 0.9× slower over CPU, due to the memory transfer overhead not accounted for in kernel runtime. In addition to the above works, acceleration of statistical STA (SSTA) is also attempted using GPUs and FPGAs, which is a different scope of discussion [24], [25], [26].

To accurately address the runtime bottleneck of a real STA run, we have profiled the widely-adopted open-source STA engine OpenTimer [3], [5]. As OpenTimer is reported to outperform commercial STA tools [3] in terms of speed, we regard its runtime footprint as a common scenario in high-performance STA engines. We draw its runtime decomposition for a full graph-based timing analysis in Figure 2. As we can observe in the figure, the RC timing step, including RC trees

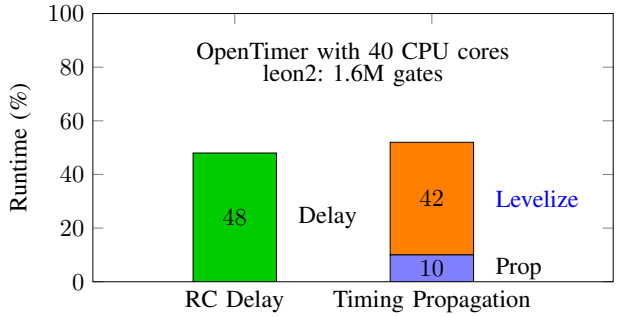


Fig. 2: The OpenTimer runtime decomposition (40 CPU cores) inside one full timing process on a million-sized design.

construction and slew parameter updates across nets, takes up a large percentage (48%) of the runtime due to the vast amount of SPEF data to process [18]. Another significant portion of runtime (42%) is taken by constructing levelized task dependency graphs with a size proportional to the circuit size, on which timing propagation is conducted. Handling the signal relations between pins is a common burden in all STA engines, due to its difficulty to be parallelized.

B. Multi-corner Timing Analysis in Advanced Nodes

With the continued increase of design complexity and advancements in technology nodes, the timing behavior of a circuit design is more and more involved with variations introduced across design process, manufacturing noise, and operating conditions 3. Therefore, *multi-corner* timing analysis is proposed by repeatedly applying the STA engine on different cell libraries characterized under different PVT conditions. The number of such combinations is often tens or hundreds to cover a large enough set of scenarios during signoff, effectively magnifying the STA runtime by 10× or even 100×.

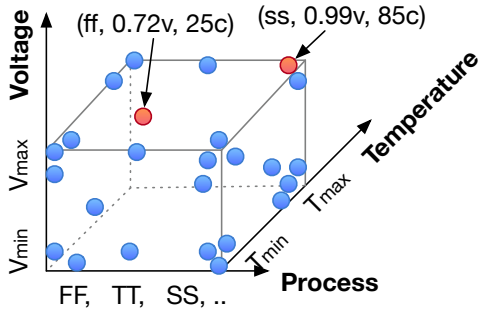


Fig. 3: PVT corners are combinations of process, voltage, and temperature parameters. They can be visualized as points in a 3-dimensional cube.

To reduce the runtime of the expensive multi-corner analysis, researchers have proposed a number of approaches recently [12], [13], [14], [15], [16], [17]. One direction is to apply branch-and-bound during corner parameter selection and STA. For example, [17] proposes to prune the search space of multi-corner exploration with delay upper-bound estimations. This is further improved by [14] which runs different branch-and-bound algorithms in parallel by CPU-based multi-threading. Another direction is to approximate the corners by incorporating prior knowledge like the clock tree update in hold analysis [13] or the local linearity between similar corners and modes [16], [15]. One recent direction applies machine learning (ML) to multi-corner STA by predicting unobserved corners given observed ones [12]. However, they still suffer from about 10% maximum error due to the inherent stability and explainability problem of ML models.

Despite extensive research on multi-corner analysis, most of them are limited to CPU-based parallelism. Worse still, they introduce accuracy loss in the multi-corner analysis results in exchange for speed. Such accuracy loss can be significant in more advanced nodes due to highly nonlinear timing effects and more complex timing models. Since the circuit structure does not change across different corners, a large amount of data parallelism remains unused.

C. GPU-Accelerated STA Challenges

Hybrid compute systems with heterogeneous computing resources like CPUs and GPUs are becoming increasingly prevalent. Contrary to a CPU consisting of a few high-performance big cores, a GPU is made up of thousands of tiny cores arranged into streaming multiprocessors. It attempts to achieve a high throughput through extensive parallelization while minimizing threading costs. For instance, the RC timing computation for different nets can be done independently because the RC delay and slew parameters can be isolated [18]. Furthermore, the GPU's compute capability can also be used to sort out dependent tasks by computing a topological order in parallel. We highlight below 3 challenges for speeding up these STA tasks.

1) *Irregular computational patterns*: nearly all STA tasks contain irregular computation patterns, including recursive procedures, dynamic data structures, and graph traversal.
2) *Frequent memory access*: even though the RC delay computation for different nets are independent, each of them requires randomly accessing GBs of memory on million-gate designs, due to detailed parasitics and various local RC tree structures.
3) *Large multi-corner memory footprint*: in multi-corner analysis, memory access problems are worsened by an additional 10–100×, due to memory footprint proportional to the number of corners. This gives even more pressure on memory and cache management.

The above challenges require very strategic data structure models and decomposition algorithms to obtain a reasonable runtime speed-up.

III. Algorithm

In this section, we present our GPU acceleration algorithm details. Our overall taskflow is presented in Figure 4, where each arrow indicates a task dependency. There are 3 basic steps in our taskflow: *RC delay computation*, *timing propagation*, and *levelization*. The timing propagation step is further

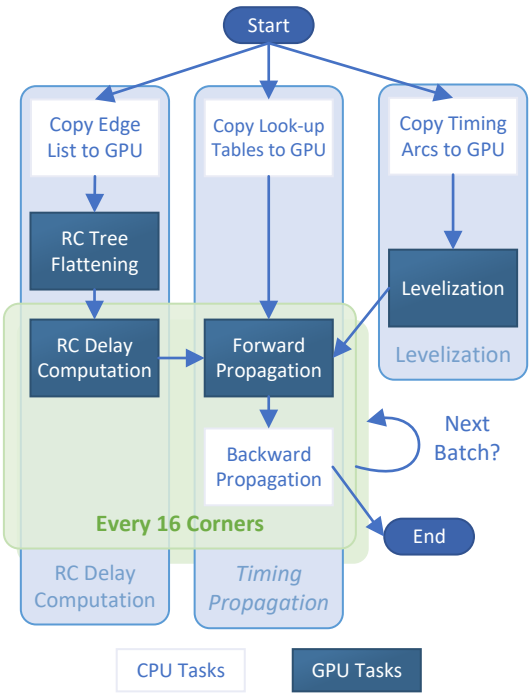


Fig. 4: Overall taskflow of our GPU-accelerated multi-corner STA engine. The green frame captures the tasks running in parallel for a batch of corners.

decomposed to *forward propagation* and *backward propagation*. In Figure 4, dark color indicates GPU-accelerated steps, including *RC delay computation*, *RC tree flattening*, *forward propagation*, and *levelization*. Because *backward propagation* has a much lower workload and nearly negligible runtime, we leave it on CPU. We focus on using GPU to address the major scalability issues and runtime bottlenecks shown in Figure 2.

A. RC Delay Computation

A majority of STA runtime is taken by the RC delay computation step [18]. We begin by analyzing the model and the equations for calculating the delay and slew parameters through an RC network. Then we demonstrate how to build GPU-friendly data structures and algorithms. We adopt a variant of the Elmore delay model [18] to approximate interconnect delay, which had been widely adopted by many different STA engines [3], [8], [11]. We approximate the interconnect delay based on an Elmore delay model variant [18] that had been implemented by many STA engines [3], [8], [11]. As illustrated in Figure 5, our goal is to calculate the impulse and delay between the source pin (*Port*) and each sink pin (*Taps*).

CPU Implementation [3]. Dynamic programming (DP) is a standard approach to implementing RC delay calculation. There are four stages in this algorithm:

1) Compute the pin load (i.e. the lumped capacitance) for each node u, denoted as $load_u$.

$$\begin{aligned} load_A &= Cap_A + Cap_B + Cap_C + Cap_D \\ &= Cap_A + load_B + load_D. \end{aligned} \tag{1}$$

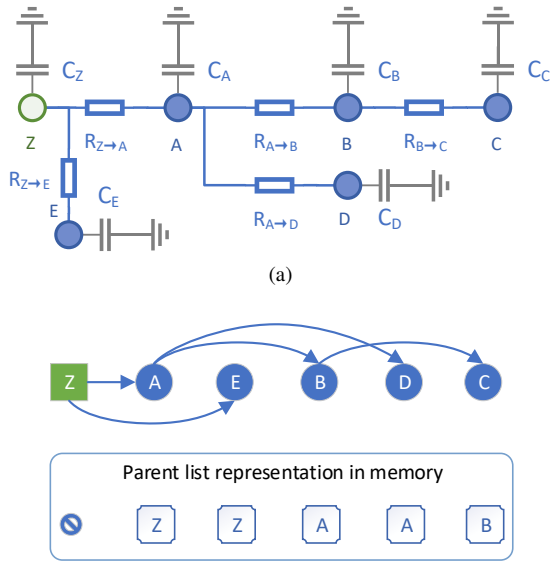


Fig. 5: Example of a net RC tree model with parasitics. (a) The edge resistance and node capacitance on parasitic RC tree. (b) The BFS order of nodes represented as a 1D flattened array with a list of parent indices.

2) Compute the delay between *Port* and u, denoted as $delay_u$.

$$delay_u = \sum_{v \in nodes} Cap_v R_{Port \to \mathrm{LCA}(u,v)}, \tag{2}$$

where LCA denotes lowest common ancestor.

$$\begin{aligned} delay_B =& R_{Z\to A} Cap_A + R_{Z\to A} Cap_D + \\ & (R_{Z\to A} + R_{A\to B}) Cap_B + \\ & (R_{Z\to A} + R_{A\to B}) Cap_C \\ =& delay_A + R_{A\to B} load_B. \end{aligned} \tag{3}$$

3) Calculate the sum of capacitance and delay products in subtrees of u, denoted as $ldelay_u$, similar to step 1.
4) Calculate the beta and impulse parameters between the source *Port* and the sinks u, based on the $ldelay$ of nodes, similar to step 2.

On CPU implementations, each parameter is typically computed using several passes of RC tree traversals. This yields a linear runtime complexity proportional to the tree size, although it may not be GPU-efficient due to its irregular recursions. Therefore, in our GPU implementation, we choose to use *breadth-first search* (BFS) traversals. In our BFS implementation instead, we precompute once a node order for each RC tree. This order ensures that every parent node is before any of its children. In other words, a tree edge $u \to v$ makes u appear before v in the BFS order, as illustrated in Figure 5. The BFS order represents the structure of an RC tree concisely and efficiently for GPU execution. All we need to do is to visit the tree nodes forwardly or backwardly via the ordered sequence, according to the DP update directions.

Algorithm 1 is our algorithm for GPU-accelerated RC tree flattening. It computes the node BFS order of one net using an

Algorithm 1: Flatten RC Trees

Input: N as #nets, (M, E) as (#nodes, #es) in all nets
Input: $roots[0..N-1]$, the root indices of each net
Input: $es[0..E-1]$, the undirected edge $\{(a, b)\}$
Input: $ndstart[0..N]$, the offsets of each net in node arrays, with $ndstart[N] = M$
Input: $estart[0..N]$, the offsets of each net in edge arrays, with $estart[N] = E$
Input: $dis[0..M] = \infty$, $cnts[0..M] = 0$
Output: $order[0..M-1]$, the BFS order for each net

```
/* Process one net w/ blockDim.x threads */
1  netID = blockIdx.x;                          ▷ gridDim.x = #nets
2  threadID = threadIdx.x;                      ▷ blockDim.x = 64
3  nst = ndstart[netID];                        ▷ start node offset
4  nend = ndstart[netID + 1];                   ▷ end node offset
5  est = estart[netID];                         ▷ start edge offset
6  eend = estart[netID + 1];                    ▷ end edge offset
7  dis[nst + roots[netID]] = 0;
8  for d = 0, 1, 2, ..., (nend − nst) do
9      for i = est + threadID to eend step blockDim.x do
10         (a, b) = edgelist[i];
11         if dis[a] == d and dis[b] >d + 1 then
12             dis[b] = d + 1;
13             atomicAdd(cnts[d], 1);
14         end
15         else if dis[b] == d and dis[a] >d + 1 then
16             dis[a] = d + 1;
17             atomicAdd(cnts[d], 1);
18         end
19     end
20     __syncthreads();
21     break when cnts[d] == 0;
22 end
23 countingSort(dis, cnts, order, threadID);
```

Algorithm 2: Compute RC Delay for Corner Batches

Input: N as #nets, M as #nodes in all nets, BC as the batch size of multi-corner settings
Input: $st[0..N]$, the offsets of each net in arrays of nodes
Input: $parent[0..M-1]$, the index of parent of every nodes
Input: $resp[0..M-1]$, the resistance between nodes and their parent
Input: $cap[0..4M-1][BC]$, the capacitance of nodes, each in 4 different combinations
Output: $load[0..4M-1][BC]$, $delay[0..4M-1][BC]$, $impulse[0..4M-1][BC]$: arrays of results of load, delay and impulse, respectively

```
1  net = blockIdx.x × blockDim.x + threadIdx.x;
2  cond = threadIdx.y;
3  corner = threadIdx.z;
4  if net ≥ N then return;
5  offsetL = st[net];                           ▷ node offset start
6  offsetR = st[net + 1];                       ▷ node offset end
7  Initialize load, delay, ldelay to zero;
8  Initialize β = 0 as an auxiliary array;
9  for j = offsetR − 1 down to offsetL do
10     load[4j + cond][corner] += cap[4j + cond][corner];
11     load[4parent[j] + cond][corner] +=
         load[4j + cond][corner];
12 end
13 for j = offsetL + 1 to offsetR − 1 do
14     t = load[4j + cond][corner] × resp[j];
15     delay[4j + cond][corner] =
         delay[4parent[j] + cond][corner] + t;
16 end
17 for j = offsetR − 1 down to offsetL do
18     ldelay[4j + cond][corner] +=
         cap[4j + cond][corner] × delay[4j + cond][corner];
19     ldelay[4parent[j] + cond][corner] +=
         load[4j + cond][corner];
20 end
21 for j = offsetL + 1 to offsetR − 1 do
22     t′ = ldelay[4j + cond][corner] × resp[j][corner];
23     β[4j + cond][corner] =
         β[4parent[j] + cond][corner] + t′;
24     impulse[4j + cond][corner] =
         2β[4j + cond][corner] − delay[4j + cond][corner]²;
25 end
```

input edge list in two stages: (1) computing distances to root for each node, and (2) sorting nodes by their root distances. The time complexity of the first step is $\mathcal{O}(n^2)$ where n denotes the net size. Because of the limited net size (usually less than a few hundred), such an $\mathcal{O}(n^2)$ algorithm is efficient enough and can be even better parallelized on GPU due to its simplicity. A block of 64 threads are launched for each net to process its edge list. The edge list would be traversed multiple times to compute the order. In each iteration (lines 9-19), we obtain a new batch of nodes with the same root distance. Finally, we sort all nodes by their root distance using a GPU parallel counting sort with $\mathcal{O}(n)$ time complexity.

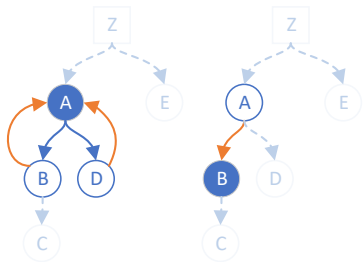
Based on the computed BFS order, the pseudocode for our RC computation GPU kernel is shown in Algorithm 2. We launch one kernel thread for each unique combination of Early/Late, Rise/Fall, net index, and corner index in a corner batch. The details of multi-corner parallelism are introduced later in Section III-E. Firstly, the *netID* and *condID* are computed on lines 1-4. We compute the net data offsets in parameter arrays on lines 5-6 and fill the output arrays with initial zero values on lines 7-8. After the initialization, we traverse and calculate the RC parameters $load$ (lines 9-12), $delay$ (lines 13-16), $ldelay$ (lines 17-20), $beta$ and $impulse$ (lines 21-25).

Our algorithm works on our optimized RC tree data structure on GPU memory, where we store parent indices in the parent array for all RC tree nodes (Figure 5(b)). This concise representation of parent-child relations on GPU ensures a balanced workload during different DP update passes. We take the recursive equation of $load$ as an example,

$$load_u = cap_u + \sum_{v \in \{\text{children of } u\}} load_v, \tag{4}$$

as also illustrated in Figure 6(a). Because we only keep parent indices instead of a large adjacency list, we cannot enumerate all children of the node u and compute the sum as the equation requires. Instead, we equivalently regard the $load_u$ as running sums. Algorithm 2 amend the running sum at u on all children of u (lines 11), progressively arriving at the final result. Because the children of u appear after u in the BFS order, and because we scan the BFS order from backward, we would already have processed all children of u before encountering u in the sequence.

Another example is the recursive equation of $delay$, which



(a) Upward (b) Downward

Fig. 6: Two different DP directions. (a) Upward recursive update, where the value of children needs to be computed before the value of parents. (b) Downward recursive update, where the parent values are computed before children.

has a different direction:

$$delay_v = delay_u + pres_v \times load_v \tag{5}$$

as shown in Figure 6(b). Here u denotes the parent of v. This equation is straightforward to implement using our array of parent indices and a forward BFS order scan (lines 14-15). The updates of $ldelay$ and $beta$ share similar patterns with $load$ and $delay$ and can be done analogously.

The bottleneck of RC computation is irregular global memory access with a large number of nets and independent analysis combinations. To address this, we design a data structure that is friendly for global memory access. We optimize memory bandwidth usage by interleaving the memory for the 4 Early/Late Rise/Fall combinations, instead of arranging them separately (Figure 7 (a) and (b)). For multi-corner analysis, we interleave different corners by assigning the corner dimension as the innermost array indices. This creates more memory coalescing capability as shown in Figure 7 (c). This arrangement ensures that adjacent 4 threads emit adjacent memory requests, which corresponds to the index equation $4i + cond$ for the i-th node and the $cond$-th combination in Algorithm 2.

B. Levelization

Levelization is an STA step that constructs level-by-level task dependencies for timing propagation tasks [3]. It takes up nearly 40% of the full timing runtime (shown in Figure 2). The inefficiency is caused by its single-threaded nature. Existing STA engines, including commercial tools like [11], perform a single-threaded DFS or BFS on the circuit logic to construct a level list and guide the parallelization of node tasks. This data structure is very time-consuming to maintain. As a result, we present a levelization algorithm with GPU acceleration.

Algorithm 3 shows our GPU-accelerated levelization process. Our key idea is to keep a node set F at the current level, called *frontiers*. Nodes that have no input edges (i.e., circuit primary input pins) become the initial frontiers (lines 1). The algorithm repeats through lines 3-6 until we have processed all nodes. On each iteration, a GPU kernel `advanceFrontiers` is invoked to find the next frontiers based on current ones in parallel.

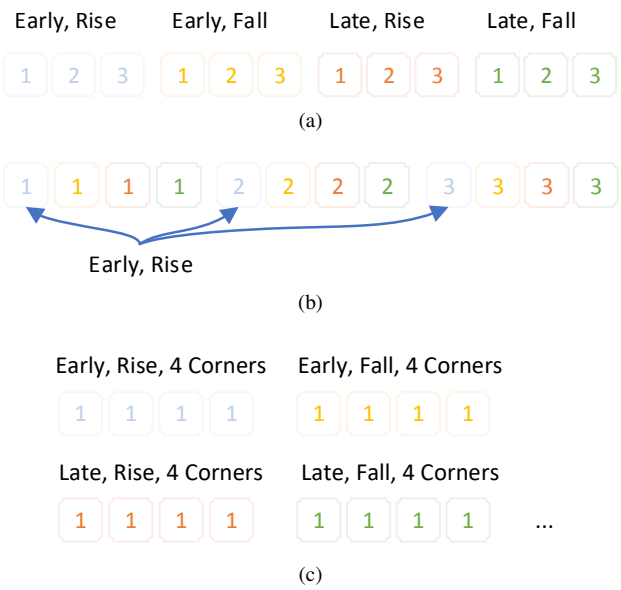


Fig. 7: Memory arrangement for Early/Late and Rise/Fall cases. (a) Independent access; (b) Interleaved access; (c) Interleaved access with multi-corner optimization.

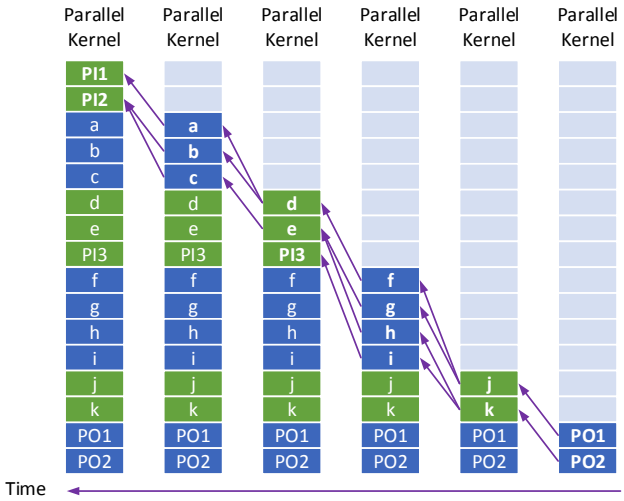


Fig. 8: The resulting level list for the circuit graph in Figure 1, as well as the levelization process on GPU. Node names in bold indicate frontiers at the current iteration.

The `advanceFrontiers` procedure accepts a list of current frontiers and launches one thread to process a single frontier. The output edges of frontiers (lines 11-16) are enumerated. We decrement the in-degree of a node v by one for each output edge pointing to it. We push v to the next set of frontiers once its in-degree drops to zero after a decrement.

In this algorithm, the edge explorations of different frontiers are performed simultaneously, while the output edges from one frontier are processed one by one. The workload among GPU threads is proportional to the out-degree of nodes, which can be very imbalanced. We have adopted a *reverse* technique to tackle this problem by observing that the in-degrees of nodes are generally smaller and much more balanced. For example, in netcard [18] with 1.5M of gates, the maximum out-degree

Algorithm 3: Levelization

Input: $nodes$, the set of graph nodes
Data: the current in-degree in and the adjacency list out
Output: a node level list

$F \leftarrow \{f \in nodes : in_f = 0\}$;
while F *is not empty* **do**
  output F;
  $F' \leftarrow \{\}$;
  Call `advanceFrontiers` on F and get F';
  $F \leftarrow F'$;
end
Kernel Function `advanceFrontier`:
  Input: the old frontier F
  Data: the adjacency list out, in-degree array in
  Output: the new frontier F'
  $nodeID \leftarrow$ **blockIdx**.x $\times$ **blockDim**.x + **threadIdx**.x;
  if $nodeID \geq$ size(F) **then return**;
  for v *in* $out[nodeID]$ **do**
    $oldvalue \leftarrow$ `atomicAdd(`$in[v]$, *-1*`)`;
    if $oldvalue$ = *1* **then**
      Add v to F';
    end
  end
  return G;

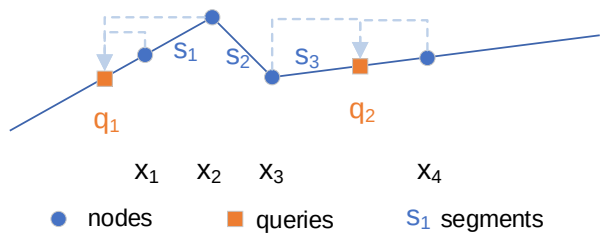


Fig. 9: Example of 1D lookup table query. This 1D lookup table is essentially a piecewise linear function with three segments s_1, s_2, s_3. There are two queries q_1 (that hits s_1) and q_2 (that hits s_3). We get the result by evaluating the queried x-value on the specific segment, regardless of interpolation (like q_2) or extrapolation (like q_1).

and in-degree are 260 and 8, respectively. We reverse the edge directions before running the levelization on the graph, which gives higher parallelism during the edge exploration. After the levelization, we can retrieve the level orders of the original graph by reversing back the level orders, as illustrated in Figure 8. The level list of large designs typically gives thousands of independent node tasks in each level, leading to enough parallelism for propagation.

C. Timing Propagation and LUT

According to the runtime decomposition in Figure 2, the timing propagation step is efficient on CPU because of the small LUT size. Despite this, we managed to obtain a modest speed-up, especially for million-gate circuit designs, by migrating it to GPU. In NLDM model, the delay and slew for cell arcs are modeled by a piecewise linear 2D function with input slew and output capacitance as its inputs. This function is characterized by around 7×7 sample points obtained from circuit simulations and queried by bilinear interpolation.

Algorithm 4: Multi-Corner LUT Interpolation

`/* Input: line` (x_1, y_1)--(x_2, y_2) `*/`
`/* Input: the` x `value queried */`
1 **Function** `interpolate(`x_1, x_2, y_1, y_2, x`)`:
2 **if** $x_1 = x_2$ **then return** y_1;
3 **else return** $d_1 + (d_2 - d_1)\frac{x - x_1}{x_2 - x_1}$;
4 **end**
`/* Input:` $n \times m$ `look-up table with corner batch` BC `*/`
`/* Input: the point queried` (x, y) `and the corner index` $corner$ `*/`
5 **Function** `lut_lookup(`$n, m, X, Y, mat, x, y, corner$`)`:
6 $i' \leftarrow 0$;
7 $i \leftarrow \min(1, n-1)$;
8 **while** $i + 1 < n$ *and* $X[i] \leq x$ **do**
9 $i' \leftarrow i$;
10 $i \leftarrow i + 1$;
11 **end**
12 $j' \leftarrow 0$;
13 $j \leftarrow \min(1, m-1)$;
14 **while** $j + 1 < m$ *and* $Y[j] \leq y$ **do**
15 $j' \leftarrow j$;
16 $j \leftarrow j + 1$;
17 **end**
18 $r_{i'} \leftarrow$`interpolate(`$Y[j']$, $Y[j]$, $mat[i', j'][corner]$, $mat[i', j][corner]$`)`;
19 $r_i \leftarrow$`interpolate(`$Y[j']$, $Y[j]$, $mat[i, j'][corner]$, $mat[i, j][corner]$`)`;
20 $r \leftarrow$`interpolate(`$X[i'], X[i], r_{i'}, r_i$`)`;
21 **return** r;
22 **end**

We present our GPU-accelerated LUT table lookup algorithm in Algorithm 4. We calculate 2D bilinear interpolations using three 1D linear interpolations of sample points (lines 18-20). Given a single x, each 1D linear interpolation finds the y of a piecewise linear polyline at x. When the given x exceeds the range of sample points, an extrapolation is performed instead of interpolation, which introduces a branch divergence on GPUs. To this end, we generalize the process to cover both extrapolation and interpolation under the same code, as illustrated in Figure 9. The idea is to locate the line segment (or half-line) where x locates and then use a unified equation to solve y. We deal with the cases where LUT degenerated to a row, a column, or a single value, by setting $i' = i$ in these cases. A linear search is performed to find these indices because of the small size of LUTs.

D. Device-to-device Data Bridge between STA Steps

During the STA process, consecutive STA steps need to exchange intermediate timing analysis data. For example, the RC delay step computes net delay, capacitive load, and impulse, which are used in propagation to compute the signal arrival time. However, such communication is difficult to handle due to the data-structural difference between flattened RC trees and levelized arc tables. In STA engines like OpenTimer [3], a CPU is used to "translate" the timing data between different structures. In the multi-corner case, this is no longer scalable due to the data size proportional to the number of corners.

To solve the above problem, we present Algorithm 5 as our device-to-device data transfer algorithm that bridges the

gap between RC delay computation and timing forward propagation. The algorithm makes use of both CPU and GPU. On CPU, the algorithm preprocesses a mapping of memory offsets between flattened RC trees and the flattened arc table (lines 1–10). The mapping itself is small in size because it is set up only once for every single net arc, regardless of the number of corners and signal rise/fall conditions. After the mapping is ready, the algorithm copies it to GPU (line 11), where it is used to move groups of timing data to their destined locations (lines 12–18).

Algorithm 5: Device-to-device Data Transfer

```
   /* CPU code                                                    */
1  arcid2flatpos ← [];
2  for every net i do
3      r ← driver pin of net i;
4      L, R ← the range of net i's flattened RC storage;
5      for every sink pin p in net i do
6          t ← the index of arc r → p;
7          t' ← the position of p in [L, R];
8          arcid2flatpos[t] ← t';
9      end
10 end
11 copy arcid2flatpos to GPU;
   /* GPU code                                                    */
12 delay, impulse ← the GPU RC delay/impulse array;
13 arcdelay, arcimpulse ← the GPU flattened arc table;
14 arcid ← blockIdx.x × blockDim.x + threadIdx.x;
15 elrf ← threadIdx.y;
16 corner ← threadIdx.z;
17 arcdelay[arcid, elrf, corner] ← delay[arcid2flatpos[arcid],
     elrf, corner];
18 arcimpulse[arcid, elrf, corner] ←
     impulse[arcid2flatpos[arcid], elrf, corner];
```

E. Multi-corner GPU Parallelization

For each PVT condition under multi-corner STA, the STA engine computes the circuit delay and slack using a unique cell library with its own set of pin loads and LUTs. Parallelization between corners can exceed tens or even hundreds, with following properties:

1) While pin loads and LUTs are re-modeled, the circuit topology remains unchanged across different corners. The topology-related computation (including RC tree flattening in Section III-A and levelization in Section III-B) can thus be cached and reused.
2) The computation across different corners have similar patterns and almost no branch divergence. This leads to efficient GPU-friendly single-instruction-multiple-thread (SIMT) behavior.

Fully utilizing these properties, we develop a GPU-accelerated multi-corner STA flow. Storing all intermediate results for hundreds of corners is impractical on the limited GPU memory. Moreover, the thread block size proportional to the number of concurrent corners poses inefficient restrictions on GPU thread block scheduling. As such, we split the PVT corners into equal-sized batches with BC corners each, and compute the batches iteratively. In order to maximize data parallelism and SIMT performance, we put the corner iteration into the most inner loop, i.e. the last index of the 3-dimensional CUDA thread group. Meanwhile, we also arrange the memory structure similar to Figure 7 so that memory requests from consecutive corners are interleaved and thus coalesced.

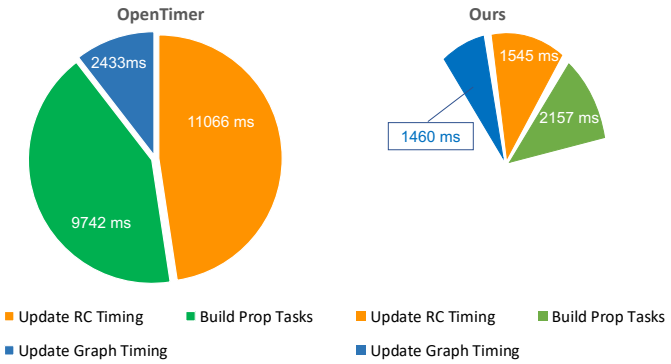


Fig. 10: Runtime breakdown of the circuit leon2 (21M nodes).

IV. Experimental Results

We implemented our GPU-accelerated STA algorithm on top of OpenTimer [3] and evaluated the results using TAU15 contest benchmarks [18]. We re-synthesized all the benchmark netlists under an industrial 14nm technology. This gives realistic multi-corner cell libraries under different operating conditions. We do not compare with commercial tools (e.g., PrimeTime, OpenSTA) because they do not support GPU. Also, such a comparison may not be fair because of different application scopes. All experiments are undertaken on an Ubuntu Linux machine with 40 CPU cores at 2.10 GHz, 512 GB RAM, and 1 Nvidia A40 GPU. We configured the kernel execution with about 128 threads per block for all GPU kernels. Our algorithms are implemented using the parallel task programming framework, Taskflow [5], [27] to schedule CPU-GPU dependent tasks. We measured the *end-to-end* STA runtime including both GPU kernels and memory preparation operations.

A. Single-corner Full Timing

Table I lists the benchmark statistics and the overall performance comparison between our approach and OpenTimer. We measure the runtime to complete one iteration of full-timing update on 15 benchmarks. The netlists of these benchmarks were used in TAU15 Contest to evaluate contestants' entries at a large scale. The gates of these netlists are re-mapped to a new cell library under 14nm technology. We ran both OpenTimer and our algorithm using the maximum hardware concurrency of 16 CPUs and 1 GPU on our platform. Our runtime is faster than OpenTimer across all but the smallest benchmarks. The three largest speed-up values we observed are 4.05× on netcard (1.5M gates), 4.07× on leon2 (1.6M gates), and 3.51× on leon3mp (1.2M gates). The speed-up values become remarkable at large designs when generated STA graphs contain tens of millions of nodes and edges.

Figure 10 shows the runtime breakdown of OpenTimer and our algorithm for notable items (>1000 ms) on the largest

TABLE I: Performance comparison between OpenTimer (16 CPUs) and our GPU-accelerated implementation (1 GPU) to complete one iteration of full timing on large designs (>10K gates) of TAU 2015 contest benchmarks under 14nm technology.

Benchmark	# PIs	# POs	# Gates	# Nets	# Pins	# Nodes	# Edges	OpenTimer Runtime (16 CPUs)	Our Runtime [28] (16 CPUs 1 GPU)	
									Runtime	Speed-up
aes_core_14nm	260	129	22938	23199	66221	413058	499688	276 ms	283 ms	0.98×
vga_lcd_14nm	89	109	139529	139635	380730	1949332	2636815	1368 ms	659 ms	2.08×
vga_lcd_iccad_14nm	85	99	259067	259152	662179	3539206	4234464	2612 ms	951 ms	2.75×
b19_14nm	22	25	255278	255300	776320	4416480	5623578	3520 ms	1155 ms	3.05×
cordic_ispd_14nm	34	64	45359	45393	127993	730590	910649	508 ms	369 ms	1.38×
des_perf_ispd_14nm	234	140	138878	139112	371587	2095933	2473864	1679 ms	649 ms	2.59×
edit_dist_ispd_14nm	2562	12	147650	150212	416609	2555873	3562491	2056 ms	799 ms	2.57×
fft_ispd_14nm	1026	1984	38158	39184	116139	631491	868498	457 ms	353 ms	1.29×
leon2_14nm	615	85	1616369	1616984	4178874	22450936	28114268	23928 ms	5879 ms	4.07×
leon3mp_14nm	254	79	1247725	1247979	3267993	17647115	22807349	18174 ms	5174 ms	3.51×
netcard_14nm	1836	10	1496719	1498555	3901343	21023425	25014009	21320 ms	5259 ms	4.05×
mgc_edit_dist_14nm	2562	12	161692	164254	444693	2431266	3355118	1913 ms	793 ms	2.41×
mgc_matrix_mult_14nm	3202	1600	171282	174484	489670	2710343	3415291	1906 ms	798 ms	2.39×
pci_bridge32_14nm	160	201	40790	40950	108172	577083	696170	404 ms	318 ms	1.27×
tip_master_14nm	778	857	37715	38493	95524	533690	602224	341 ms	338 ms	1.01×

 # PIs: number of primary inputs **# POs**: number of primary outputs **# Gates**: number of gates **# Nets**: number of nets
 # Pins: number of pins **# Nodes**: number of nodes in the STA graph **# Edges**: number of edges in the STA graph

benchmark, leon2. OpenTimer spends 9742 ms to sort out the pin dependency, due to its unavoidable overhead on additional data structures and sequential nature. In our implementation, we use GPU to levelize the graph and run multiple tasks (e.g., update RC timing) in a single batch. We do not need as many tasks as OpenTimer but a single kernel to establish the topological dependency, which leads to just 2157 ms runtime. We observe a large amount of runtime reduction from updating RC timing. It takes 11066 ms for OpenTimer to finish RC timing whereas we reach the goal by 7.16× faster. Our runtime for updating the graph timing is a bit faster (1460 ms vs 2433 ms), due to our GPU-based LUT interpolation.

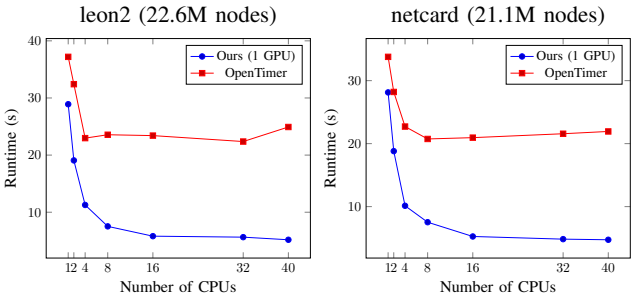


Fig. 11: Runtime values at different numbers of CPUs. Our runtime under 2 CPUs and 1 GPU is faster than OpenTimer of 16–40 CPUs.

Figure 11 draws the runtime scalability versus increasing numbers of CPUs on the two largest designs, leon2 and netcard. Increasing the number of CPUs can speed up our overlapped CPU-GPU tasks with faster data transforms. We observe both methods scale up to 10 CPUs. Regardless of CPU numbers, our runtime is always faster than OpenTimer, and there exists a remarkable gap. The largest speed-up occurs at 40 CPUs, where ours is faster than OpenTimer by 4.81× on leon2. These results clearly demonstrate the strength of our approach that unleashes the computing power of GPUs beyond the limitation of CPU-based parallelism.

B. Single-corner Incremental Timing

The success of GPU acceleration relies on a large enough data size and computation, which is abundant in the case of full timing updates on STA graph. During incremental timing, computation varies and may scope to a small local region or the entire timing landscape. Pins in this region are called *propagation candidates* which are the union of fan-in and fan-out cones spanned by frontier pins in incremental timing [3]. Considering the distinct performance characteristics between CPU and GPU, the most effective approach to incremental timing is a mixed strategy. When the number of propagation candidates is large, we use GPU; or we fall back to the existing CPU version of OpenTimer when propagation candidates are scarce.

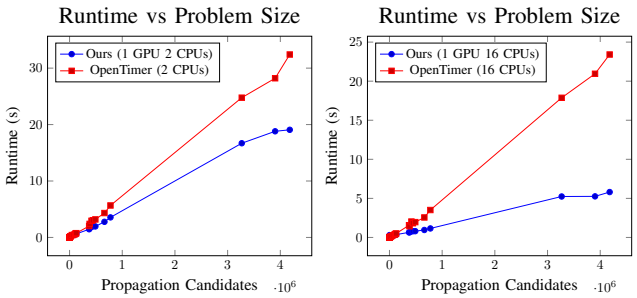


Fig. 12: Runtime values at different problem sizes. Beyond about 60K propagation candidates, our runtime is always faster than OpenTimer at any CPU numbers.

Figure 12, 13, 14 compares runtime at different sizes of problem candidates, nets, and gates, respectively, between our GPU algorithm and OpenTimer under different CPU concurrency. For problem size smaller than 10K, we run slower than OpenTimer but the runtime difference is negligible ($<$ 80 ms). Beyond the threshold of 67K propagation candidates, our runtime is always faster than OpenTimer. The performance margin becomes bigger as we increase the problem size. In terms of the number of nets, the threshold is about 40K nets.

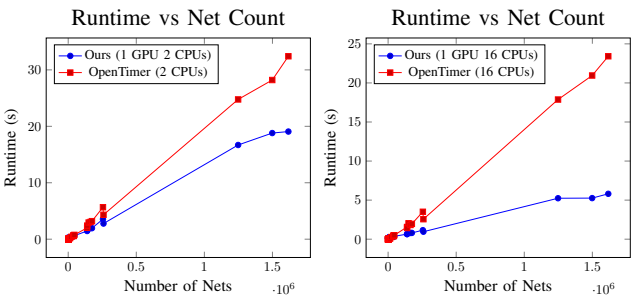


Fig. 13: Runtime values at different net counts. Beyond about 40K nets, our GPU-accelerated RC computation is always faster than OpenTimer, regardless of CPU numbers.

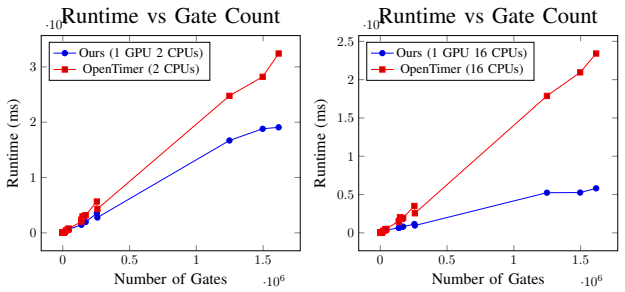


Fig. 14: Runtime values at different numbers of gates (∼LUT numbers). Beyond about 45K gates, our GPU-accelerated LUT interpolation becomes faster than OpenTimer.

We observe little benefit at small net counts due to the data and kernel overheads, but we are consistently faster at larger net counts. The threshold of gate numbers is roughly 45K, which corresponds to 360K LUTs. As LUT interpolation is less data- and compute-intensive than other tasks, the performance margin is expected to become closer with increasing number of CPUs. To sum up, the performance benefits of our GPU-accelerated STA algorithm are remarkable when applications define large numbers of propagation candidates, for example, timing-driven placement and routing [29], [30].

C. Multi-corner Analysis

In this section, we present our results on multi-corner STA acceleration. Our industrial 14nm technology includes a diverse range of cell libraries under voltages ranging in [0.66, 0.99], temporatures ranging in [-40c, 125c], and 3 different process corners (ff, ss, tt). We choose 128 corners from all available libraries for testing. Table II shows a detailed runtime comparison between our single-corner and multi-corner analysis, with multi-corner batch size set to $BC = 2, 4, 8, 12$, and 16. Despite both running on GPU, our multi-corner algorithm outperforms our original single-corner algorithm by a large amount. On large designs like leon2, leon3mp, and netcard, we can achieve 3.85×–3.90× speed-up by computing 4 corners in parallel, compared to computing corners one by one. A larger batch size gives better performance. By computing 16 corners in parallel, the speed-up on leon2, leon3mp, and netcard is enlarged to 5.66×–6.34×. These results have proven the efficiency of our GPU-accelerated multi-corner STA algorithm on exploring data parallelism across corners.

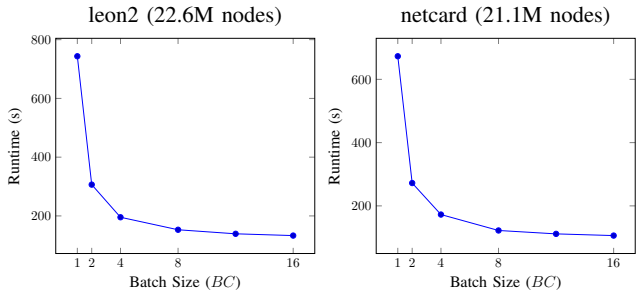


Fig. 15: Runtime values at different corner batch sizes (BC) for analyzing 128 corners. The data point at $BC = 1$ comes from our GPU-accelerated single-corner STA engine, which is our conference version [28]. Other data points are collected using our multi-corner STA engine.

Figure 15 visualizes the runtime with respect to BC on the two largest designs, leon2 and netcard. With our multi-corner acceleration techniques, a drastic speed-up can be obtained compared to the state-of-the-art GPU-accelerated single-corner STA engine ($BC = 1$ in Figure 15). A larger batch size leads to a better performance, which saturates at around $BC = 16$. Note that regarding data parallelism, a batch size of 16 already fulfills the half-warp SIMT dispatching scheme of current CUDA architecture and can eliminate branch divergence completely.

Note that when compared with the original CPU-based OpenTimer, a speed-up ratio in Table II should be multiplied with the GPU acceleration speed-up in Table I, which is itself 3.51×–4.07×. This yields an overall speed-up of 22.14×–25.67× compared to running OpenTimer repeatedly for all corners. We also note that the speed-up ratio is larger for smaller designs. For example, on cordic_ispd, des_perf_ispd, and tip_master, the multi-corner speed-up ratio is more than 8×. Such counterintuitive results may come from their smaller memory footprint, due to our extensive GPU memory usage proportional to the batch size BC used.

V. Conclusion

In this paper, we have presented a new GPU-accelerated STA algorithm to go beyond the scalability of existing methods. We have developed GPU-efficient data structures and algorithms to speed up essential tasks, including levelization, delay computation, and timing propagation in updating an STA graph. We have leveraged task parallelism to describe dependent CPU-GPU tasks such that data processing and kernel computation are efficiently overlapped. We have scaled our GPU acceleration to the analysis of multiple PVT corners, which yields further runtime improvements. Compared to the state-of-the-art STA engine, OpenTimer, we achieved up to 4.07× speed-up on a large design of 1.6M gates and 1.6M nets using 1 GPU. By computing 16 corners in parallel, we achieved another 5.66× speed-up.

Our future work includes developing GPU-accelerated algorithms for different delay calculators, including current

TABLE II: Performance comparison between our single-corner STA engine [28] and our multi-corner STA engine under different multi-corner batch size $BC = 2, 4, 8, 12, 16$ to complete an 128-corner timing analysis on given designs.

Benchmark	One by One	2-way Parallel		4-way Parallel		8-way Parallel		12-way Parallel		16-way Parallel	
	Runtime	Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
aes_core_14nm	36198	9984	3.63×	6005	6.03×	4896	7.39×	4693	7.71×	4661	7.77×
vga_lcd_14nm	84369	28459	2.96×	18635	4.53×	13611	6.20×	13053	6.46×	11192	7.54×
vga_lcd_iccad_14nm	121711	43157	2.82×	28384	4.29×	20672	5.89×	18861	6.45×	18539	6.57×
b19_14nm	147849	55573	2.66×	36032	4.10×	27067	5.46×	25087	5.89×	22251	6.64×
cordic_ispd_14nm	47241	11968	3.95×	8352	5.66×	6283	7.52×	6208	7.61×	5752	8.21×
des_perf_ispd_14nm	83132	25067	3.32×	16277	5.11×	11632	7.15×	11158	7.45×	9659	8.61×
edit_dist_ispd_14nm	102255	30784	3.32×	21216	4.82×	15472	6.61×	14733	6.94×	13275	7.70×
fft_ispd_14nm	45244	10837	4.17×	7829	5.78×	6997	6.47×	6057	7.47×	6091	7.43×
leon2_14nm	752512	306496	2.46×	195424	3.85×	152859	4.92×	139099	5.41×	133053	5.66×
leon3mp_14nm	662263	250795	2.64×	171936	3.85×	122939	5.39×	110447	6.00×	105013	6.31×
netcard_14nm	673109	276992	2.43×	172576	3.90×	122229	5.51×	111485	6.04×	106096	6.34×
mgc_edit_dist_14nm	101547	34069	2.98×	20811	4.88×	15888	6.39×	14285	7.11×	13523	7.51×
mgc_matrix_mult_14nm	102153	32000	3.19×	22944	4.45×	15403	6.63×	14527	7.03×	13608	7.51×
pci_bridge32_14nm	40670	10133	4.01×	7275	5.59×	5589	7.28×	5163	7.88×	4939	8.23×
tip_master_14nm	43221	9643	4.48×	7499	5.76×	5477	7.89×	5265	8.21×	4861	8.89×

Runtime: The runtime for analyzing 128 corners in milliseconds.
One by One: Running our GPU-accelerated single corner analysis [28] for 128 times.
BC**-way Parallel**: Running our GPU-accelerated multi-corner analysis with batch size set to BC, for $128/BC$ times.

source cell delay models and reduced-order wire delay models. We also plan to incorporate GPU task parallelism using CUDA graph feature [31] to reduce the overhead of CUDA streams and enable multiple GPUs acceleration for other time-consuming STA tasks (e.g., path-based analysis [32], [33]).

References

[1] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, 1st ed. Springer Publishing Company, Incorporated, 2009.

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

[3] T. Huang, G. Guo *et al.*, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

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

[5] T.-W. Huang, C.-X. Lin *et al.*, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 974–983.

[6] T.-W. Huang, M. D. Wong *et al.*, "A distributed timing analysis framework for large designs," in *2016 53nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

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

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

[9] T.-W. Huang and M. D. Wong, "UI-timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 11, pp. 1862–1875, 2016.

[10] P.-Y. Lee, I. H.-R. Jiang, and T.-C. Chen, "Fastpass: fast timing path search for generalized timing exception handling," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2018, pp. 172–177.

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

[12] A. B. Kahng, U. Mallappa *et al.*, ""unobserved corner" prediction: Reducing timing analysis effort for faster design convergence in advanced-node design," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 168–173.

[13] S. Onaissi, F. Taraporevala *et al.*, "A fast approach for static timing analysis covering all PVT corners," in *Proceedings of the 48th Design Automation Conference on - DAC '11*. ACM, 2011, p. 777.

[14] Jing-Jia Nian, Shih-Heng Tsai, and Chung-Yang Huang, "A unified multi-corner multi-mode static timing analysis engine," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2010, pp. 669–674.

[15] S. Sripada and M. Palla, "A timing graph based approach to mode merging," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, pp. 1–6.

[16] S. Onaissi and F. N. Najm, "A linear-time approach for static timing analysis covering all process corners," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1291–1304, 2008.

[17] L. G. e Silva, L. M. Silveira, and J. R. Phillips, "Efficient computation of the worst-delay corner," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

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

[19] J. Hu, D. Sinha, and I. Keller, "TAU 2014 contest on removing common path pessimism during timing analysis," in *Proceedings of the 2014 on International Symposium on physical design*, 2014, pp. 153–160.

[20] J. Hu, S. Chen *et al.*, "TAU 2016 contest on macro modeling," in *2016 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*. ACM, 2016.

[21] T.-Y. Lai, T.-W. Huang, and M. D. Wong, "LibAbs: An efficient and accurate timing macro-modeling algorithm for large hierarchical designs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[22] P.-Y. Lee and I. H.-R. Jiang, "iTimerM: A compact and accurate timing macro model for efficient hierarchical timing analysis," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 4, pp. 1–21, 2018.

[23] H. H.-W. Wang, L. Y.-Z. Lin *et al.*, "Casta: Cuda-accelerated static timing analysis for VLSI designs," in *2014 43rd International Conference on Parallel Processing*. IEEE, 2014, pp. 192–200.

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

[25] Y. Shen and J. Hu, "GPU acceleration for PCA-based statistical static timing analysis," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 2015, pp. 674–679.

[26] J. Cong, K. Gururaj *et al.*, "Accelerating Monte Carlo based SSTA using FPGA," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 111–114.

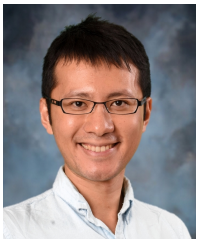
[27] T.-W. Huang, D.-L. Lin *et al.*, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 6, pp. 1303–1320, 2022.

[28] 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.

[29] Z. Guo and Y. Lin, "Differentiable-timing-driven global placement," in *ACM/IEEE Design Automation Conference (DAC)*. ACM, 2022.

[30] P. Liao, D. Guo *et al.*, "Dreamplace 4.0: Timing-driven placement with momentum-based net weighting and lagrangian-based refinement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2023.
[31] D.-L. Lin and T.-W. Huang, "Efficient gpu computation using task graph parallelism," in *Euro-Par 2021: Parallel Processing*, 2021, pp. 435–450.
[32] G. Guo, T.-W. Huang *et al.*, "Gpu-accelerated path-based timing analysis," in *ACM/IEEE Design Automation Conference (DAC)*. ACM, 2021.
[33] ——, "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.

Zizheng Guo received his B.S. degree in computer science from Peking University in 2022. He is currently a Ph.D. student in the School of Integrated Circuits at Peking University. His research interests include data structures, algorithm design, and GPU acceleration for combinatorial optimization problems. He is currently working on static timing analysis and power analysis problems in VLSI CAD. He is the First Place winner of the 2022 ACM Student Research Competition (SRC) Grand Finals.



Tsung-Wei Huang received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University (NCKU), Tainan, Taiwan, in 2010 and 2011, respectively. He obtained his Ph.D. degree in the Electrical and Computer Engineering (ECE) Department at the University of Illinois at Urbana-Champaign (UIUC) in 2017. He was an assistant professor in the ECE department at the University of Utah from 2019 to 2023, and is currently an assistant professor in the ECE department at the University of Wisconsin at Madison. Dr. Huang has been building software systems for parallel computing and timing analysis. He has received several prestigious awards to recognize his contributions, including ACM SIGDA Outstanding PhD Dissertation Award, NSF CAREER Award, Humboldt Research Fellowship Award, and ACM SIGDA Outstanding New Faculty Award.

Yibo Lin (M'19) received the B.S. degree in Microelectronics from Shanghai Jiaotong University in 2013. He obtained his Ph.D. degree in Electrical and Computer Engineering from the University of Texas at Austin in 2018. He is currently an assistant professor in the School of Integrated Circuits at Peking University. His research interests include physical design, machine learning applications, and heterogeneous computing in VLSI CAD. He is a recipient of the Best Paper Awards at premier EDA/CAD journals/conferences like TCAD, DAC, DATE, ISPD, etc.