GPU-Accelerated Static Timing Analysis

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

Tsung-Wei Huang
ECE Department
University of Utah
tsung-wei.huang@utah.edu

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

ABSTRACT

The ever-increasing power of graphics processing units (GPUs) has opened new opportunities for accelerating static timing analysis (STA) to a new milestone. Developing a CPU-GPU parallel STA engine is an extremely challenging job. We need to consider the unique problem characteristics of STA and distinct performance models between CPU and GPU, both of which require very strategic decomposition to benefit from heterogeneous parallelism. In this paper, we propose an efficient implementation for accelerating STA on a GPU. We leverage task-based approaches to decompose the STA workload into CPU-GPU dependent tasks where kernel computation and data processing overlap effectively. We develop GPU-efficient data structures and high-performance kernels to speed up various tasks of STA including levelization, delay calculation, and graph update. Our acceleration framework is flexible and adaptive. When tasks are scarce such as incremental timing, we run the normal CPU mode, and we enable GPU when tasks are massive. We have implemented our algorithms on top of OpenTimer and demonstrated promising performance speed-up on large designs. As an example, we achieved up to 3.69× speed-up on a large design of 1.6M gates and 1.6M nets using one GPU.

1 INTRODUCTION

As design complexities continue to grow larger, the need to efficiently analyze circuit timing with billions of transistors is quickly becoming the major bottleneck to the overall chip design closure process. During the timing closure, *static timing analysis* (STA) is frequently called in an inner loop of an optimization algorithm to iteratively and incrementally improve the timing of the design [1]. Optimization engines apply millions of design transforms to modify the design and the timer has to quickly and accurately update the timing to ensure slack integrity [2]. Such a process can be very time-consuming for modern very large-scale integration (VLSI) designs that incorporate billions of transistors across thousands of analysis scenarios. The efficiency of a STA engine is crucial for reasonable turnaround time and performance.

Previous work in academic research and commercial tool development has proposed various parallel STA algorithms [3–12]. Each of these algorithms has their own pros and cons, but nearly all of them are architecturally constrained by the *multithreaded* paradigm on a manycore central processing units (CPUs) platform. While results of some of those efforts have shown scalability, most are not scaling beyond 8–16 threads [12], and many of them have not been replaced with more scalable equivalents at all. With the increasing power of modern graphics processing units (GPUs), there exist new opportunities for accelerating STA to a different milestone. However, developing a high-performance STA algorithm that harnesses the power of CPU-GPU collaborative computing is an extremely challenging job. Computing a timing graph involves irregular memory access and significant diverse computational patterns, including graph-oriented computing, dynamic data structures, branch-and-bound, and recursion [4]. The resulting task graph in terms of encapsulated function calls and functional dependencies is vast and complex. Data can arrive sparsely or densely at different iterations of incremental timing. We need very strategic decomposition algorithms and data structure models to benefit from hybrid CPU-GPU computing and GPU parallelism.

As a consequence, we introduce this paper a new implementation of STA on a hybrid CPU-GPU platform. We develop GPU-efficient data structures and acceleration kernels to offload critical parts of STA computing to GPU. We have implemented our algorithms on OpenTimer, an open-source STA engine developed by Huang *et al* [4]. The core design philosophy of our algorithm is generally applicable and can be implemented for other STA frameworks. We summarize three major contributions of this work as follows:

- We leverage task parallelism to decompose the STA workload into CPU-GPU dependent tasks and enable efficient overlap between data processing and kernel computation.
- We develop GPU-efficient data structures and algorithms for delay computation, levelization, and timing propagation tasks, all of which are essential to update a STA graph.
- We develop our GPU acceleration algorithms on top of a real-world STA engine that supports incremental timing on industrial design formats. Our techniques reflect realistic performance tradeoff between CPU and GPU.

We have evaluated our algorithm on industrial designs released by TAU 2015 Timing Analysis Contest [2]. As an example, we accelerate OpenTimer by 3.69× and 3.60× on two large designs, leon2 (23M nodes and 25M edges) and netcard (21M nodes and 23M edges), using a single GPU. In the extreme, our implementation using one GPU can run faster than OpenTimer of 40 CPUs. We have also studied the effect of different factors, including problem size, net count, and gate number on the performance of incremental timing and provide guidance for when to use GPU and CPU. We believe our algorithm stands out as a unique acceleration paradigm given the ensemble of software tradeoffs and architecture decisions we have made.

The rest of this paper is organized as follows. Section 2 introduces the background of STA, GPU architecture, and formulates the problem. Section 3 presents details of our GPU-based STA algorithm. Section 4 demonstrated the experimental results of our algorithm. Finally, Section 5 concludes the paper.

2 STATIC TIMING ANALYSIS

In STA, a circuit is modeled as a *directed acyclic graph* (DAG), where each node represents a pin of a component and an edge represents a pin-to-pin connection. Figure 1 gives an example of a STA graph. The graph consists of three primary inputs, two primary outputs, and four

*Corresponding author

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-6654-2324-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415631

logic cells (1 AND gate, 1 NOT gate, and two OR gates), as well as connections (net) among components. *Graph-based* STA typically contains of two major phases, *forward propagation* and *backward propagation*, to accommodate data dependency while computing the *earliest* and the *latest* delay values at each node. Forward propagation computes quantities such as RC, slew, delay, and arrival time (AT). Backward propagation computes the values that are dependent on forward propagation, for example, required arrival time (RAT). Depending on the technology node, different delay calculators may apply. We focus on the models in the recent TAU 2014–2019 contests [2, 13, 14]. The contest benchmarks and verification scripts provide the context for experimenting new STA methods under rigorous definition of both input and output. To limit the scope of discussion, we adopt the widely used Elmore delay model [2] for net delays, and the nonlinear delay model (NLDM) for cell delays that are interpreted through 2D look-up tables (LUT) indexed by input slew and output capacitance. The criticality of a node is measured by its *slack* value, which is the difference between AT and RAT. The early and the late slacks are referred to as hold check and setup check, respectively.

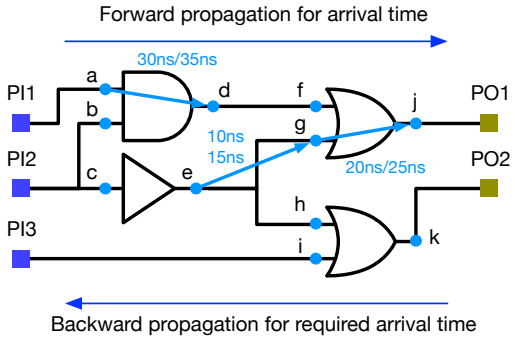


Figure 1: Static timing analysis graph of a circuit design. Each node (marked in blue) represents a pin in a component and each edge (marked in arrow) represents a pin-to-pin connection. Each delay is quantified by one min value and one max value to denote the best and worst cases.

2.1 Parallel Static Timing Analysis Engines

The ever-increasing design complexity of modern VLSI systems has put increasing strain on the efficiency of STA engines needed to analyze large designs. Computing a STA graph can involve millions or even billions of nodes and edges. The resulting task graph in terms of encapsulated function calls and data dependency is extremely vast and complex [6]. To alleviate the long runtime, recent work has explored parallel and distributed frameworks to speed up STA [3, 4, 7, 8]. For example, Huang *et al* develop a timing analysis engine, OpenTimer, using a task dependency graph to represent timing propagation tasks and their precedence [4, 6, 7]. The result has shown up to 2× improvement in runtime over loop-based OpenMP. Murray *et al* explore parallel timing propagation for FPGA designs and demonstrate 9× speedup with 32 CPU cores. Besides, recent TAU contests [2, 13, 14] have also raised new challenges on incremental timing, common path pessimism removal [9–11], and timing macro-modeling [15, 16].

Due to the threading overhead and irregular computational patterns of STA, performance of CPU-based multi-threading usually saturates at around 8–16 threads [4, 8]. To break the performance bottleneck, GPU acceleration for timing analysis is further explored [8, 17]. Wang *et al* [17] proposed acceleration techniques for the look-up table interpolation when computing the cell delays during the timing propagation, while the other steps like net delay and levelization are still on CPU. They demonstrated more than 10× speedup on the kernel propagation time over their CPU implementation. The aforementioned work from Murray *et al* [8] on FPGA designs also investigated GPU accelerated timing propagation. While they demonstrated 6.2× speedup on kernel computation time, the entire propagation runtime becomes 0.9× slower than CPU due to the data transfer overhead. In addition, there have been also attempts to accelerate Monte Carlo-based statistical STA (SSTA) algorithms with GPU and FPGA [18–20].

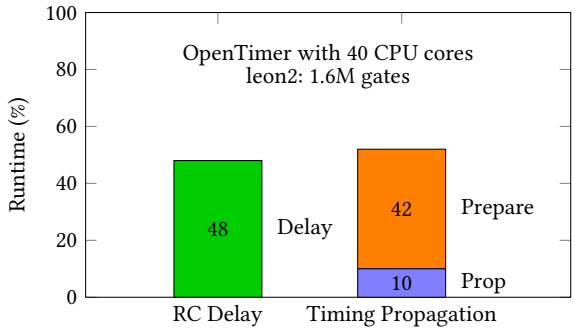


Figure 2: Runtime breakdown for OpenTimer to complete one round of full timing on a million-gate circuit using 40 CPUs.

In order to understand the runtime of each part in a STA algorithm, we have profiled the widely-used open-source STA engine, OpenTimer [4, 6]), and drawn the runtime breakdown of graph-based analysis in Figure 2. We observe a significant portion (48%) taken by RC timing, including constructing RC trees and updating parameters for computing slew and delay across nets. Updating RC timing has been always the major bottleneck in analyzing large designs due to large SPEF data we need to process [2]. Another large chunk (42%) goes to the construction of the task dependency graph to carry out the timing propagation. Since OpenTimer models each pin as a task, the resulting task graph is linearly proportionally to the size of the STA graph. Constructing a large task graph requires additional layer of data structure to represent tasks and dependencies. Also, only one thread can touch this process at a time, which becomes very time-consuming for large designs.

2.2 GPU Architecture

Heterogeneous computing systems with different compute resources are becoming popular nowadays. In general, CPUs are adopted as the host to manage and schedule all the computation tasks due to its general purpose and powerful control blocks. Data-intensive computational tasks are offloaded to GPUs for acceleration. Unlike a CPU, which has a few large "cores" with high performance, a GPU consists of streaming multiprocessors which contain thousands of less powerful small "cores". It tries to achieve high throughput with massive parallelization at low threading overhead. Therefore, a GPU is promising for plenty of small and simple tasks which may not afford the threading overhead on a CPU.

2.3 Challenges of GPU-Accelerated STA

The performance characteristics of GPU have the potential to accelerate both delay computation and timing propagation in STA. For example, computing the RC timing of each net is independent of each other as we only need to collect parameters for computing slew and delay through the generated RC tree [2]. Parasitics data is typically large (gigabytes of SPEF files) and the computation is data-driven. In addition,

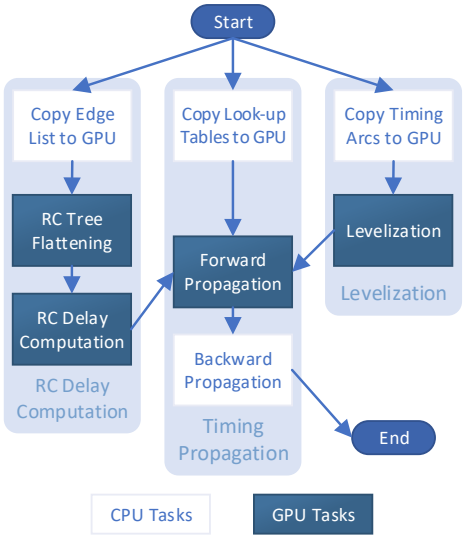


Figure 3: Overall taskflow of the GPU-accelerated STA engine.

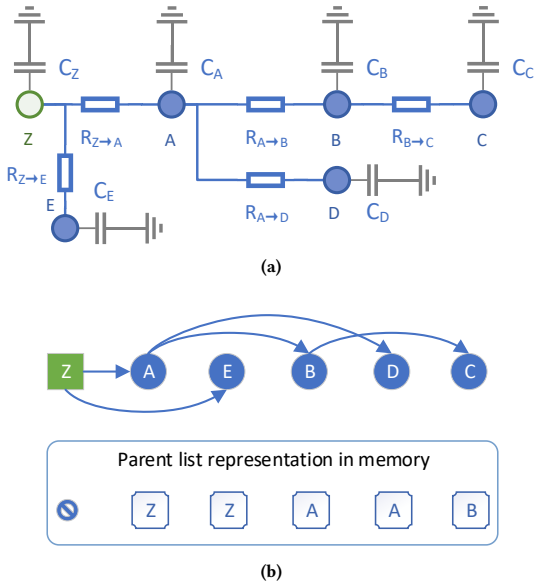


Figure 4: The parasitic RC tree model. (a) A parasitic RC tree, its node capacitance and edge resistance. (b) The above tree nodes in their BFS order, as a flattened 1D array representation, and the list of parent index.

we may leverage the power of GPU to compute the topological order of dependent tasks which would otherwise be implemented in a single-threaded graph traversal [4]. Considering the unique performance characteristics of GPU, we highlight two challenges for accelerating these tasks. (1) *Frequent memory access*: despite the independent RC delay computation of each net, it requires to access gigabytes memory to complete the computation on million-gate designs, because each net corresponds to an RC tree with the parasitics under different conditions (Early/Late, Rise/Fall); (2) *Irregular computation patterns*: many STA tasks involve irregular computational patterns, including graph traversal, dynamic data structures, and recursive procedures. Both challenges are associated with each other and require very strategic decomposition algorithms to benefit from GPU parallelism.

3 ALGORITHM

In this section, we explain the details of our GPU acceleration algorithms. The overall task graph in terms of encapsulated functions and task dependencies is shown in Figure 3, where the arrows indicate the task dependencies. We leverage the state-of-the-art parallel task programming system, Cpp-Taskflow [6], to describe CPU-GPU dependent tasks. Our task graph contains three steps: *delay computation*, *levelization*, and *timing propagation*. Timing propagation consists of *forward propagation* and *backward propagation*. We highlight the GPU-accelerated steps in dark: *RC tree flattening*, *RC delay computation*, *levelization*, and *forward propagation*. We leave *backward propagation* on CPU because its workload (i.e., update required arrival time) is much cheaper than other steps. Our goal is to address the runtime bottleneck (see Figure 2) through GPU parallelism.

3.1 RC Delay Computation

RC delay computation accounts for the majority of the runtime in most cases [2]. We first analyze the model and the algorithm for computing necessary parameters to obtain delay and slew values through a RC network. We then show how to develop GPU-efficient algorithms and data structures. We approximate the interconnect delay based on an Elmore delay model variant [2] that had been implemented by many STA engines [4, 9, 12]. As shown in Figure 4, the goal is to compute the delay and impulse between the root (*Port*) and each output pins (*Taps*).

CPU Implementation [4]. A common way to implement the RC delay computation is *dynamic programming (DP)*. This algorithm consists of four steps:

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

$$\begin{aligned} load_A =& C_A + C_B + C_C + C_D \\ =& C_A + load_B + load_D. \end{aligned} \tag{1}$$

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

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

where LCA denotes lowest common ancestor.

$$\begin{aligned} delay_B =& R_{Z\to A} load_A + R_{Z\to A} load_D + \\ & (R_{Z\to A} + R_{A\to B}) load_B + \\ & (R_{Z\to A} + R_{A\to B}) load_C \\ =& delay_A + R_{A\to B} load_B. \end{aligned} \tag{3}$$

(3) Compute the sum of the product of capacitance and delay in subtrees of u, denoted as $ldelay_u$, similar to step 1.

(4) Compute the beta and impulse value between *Port* and u, using the *ldelay* of each node, similar to step 2.

A common approach is to compute each parameter through multiple *depth-first search* (DFS) traversals of a RC tree. This results in a linear complexity proportional to the size of the tree, but it may not be efficient on GPU. Because DFS is not GPU-friendly due to recursive irregularity. Obtaining orders of traversed nodes and edges in a RC network require non-trivial memory access patterns through recursion, which are generally discouraged for GPU programming.

3.1.1 RC Tree Flattening on GPU. We use *breadth-first search* (BFS) rather than DFS. However, this requires a different approach to compute RC parameters. For each net, the DFS-based approach needs to traverse the same RC tree multiple times for resolving data dependencies between different parameter types (e.g., impulse depends on load

capacitance) under each of the four Early/Late and Rise/Fall combinations [4]. Instead, we precompute a BFS order for each tree. Based on this order, every parent appears before all its children. If there is an edge $u \to v$, then u appears before v, as shown in Figure 4. This BFS order is a GPU-efficient representation of a RC tree. We only need to traverse the nodes through the ordered sequence, either forward or backward, according to the direction of the DP step.

Algorithm 1: Flatten RC Trees

Input: N as #nets, (M, E) as (#nodes, #edges) in all nets
Input: *roots*$[0..N-1]$, the index of root of each net
Input: *edges*$[0..E-1]$, the undirected edges $\{(a, b)\}$
Input: *nodestart*$[0..N]$, the offsets of each net in arrays of nodes, with *nodestart*$[N] = M$
Input: *edgestart*$[0..N]$, the offsets of each net in arrays of edges, with *edgestart*$[N] = E$
Input: *distances*$[0..M] = \infty$, *counts*$[0..M] = 0$
Output: *order*$[0..M-1]$, nodes in 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 = nodestart[netID];                      ▷ node offset start
4  nend = nodestart[netID + 1];                 ▷ node offset end
5  est = edgestart[netID];                      ▷ edge offset start
6  eend = edgestart[netID + 1];                 ▷ edge offset end
7  distances[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 distances[a] == d and distances[b] >d + 1 then
12             distances[b] = d + 1;
13             atomicAdd(counts[d], 1);
14         end
15         else if distances[b] == d and distances[a] >d + 1 then
16             distances[a] = d + 1;
17             atomicAdd(counts[d], 1);
18         end
19     end
20     __syncthreads();                         ▷ Sync threads within a block
21     break when counts[d] == 0;
22 end
23 countingSort(distances, counts, order, threadID);
```

Algorithm 1 shows the pseudocode of our RC tree flattening algorithm. While the algorithm works on a batch of nets with each thread block processing one net, we explain it using the example of a single net. The algorithm accepts an edge list as an input and computes a BFS order of nodes. It consists of two steps: (1) compute the distances of each node to the root; (2) sort nodes according to their distance to the root. The first step adopts a parallel implementation of $\mathcal{O}(n^2)$ runtime complexity, where n is the number of nodes in a net. Since the number of nodes in a RC tree is usually around a couple hundreds, this $\mathcal{O}(n^2)$ sorting algorithm is efficient on GPU. For each net, 64 threads are launched to process the edge list of the net. We traverse the edge list multiple times. In each iteration (line 9-19), we reach a batch of new nodes with the same distance to the root of the tree. Lastly, we sort the nodes by their distance to the root using the parallel counting sort algorithm on GPU (line 23) with runtime complexity $\mathcal{O}(n)$.

Algorithm 2: Compute RC Delay

Input: N as #nets, M as #nodes in all nets
Input: *start*$[0..N]$, the offsets of each net in arrays of nodes
Input: *parent*$[0..M-1]$, the index of parent of every nodes
Input: *pres*$[0..M-1]$, the resistance between nodes and their parent
Input: *cap*$[0..4M-1]$, the capacitance of nodes, each in 4 different combinations
Output: *load*$[0..4M-1]$, *delay*$[0..4M-1]$, *impulse*$[0..4M-1]$: arrays of results of load, delay and impulse, respectively

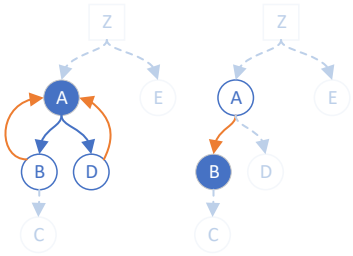
```
1  netID = blockIdx.x × blockDim.x + threadIdx.x;
2  condID = threadIdx.y;
3  if netID ≥ N then return;
4  offsetL = start[netID];                      ▷ node offset start
5  offsetR = start[netID + 1];                  ▷ node offset end
6  Initialize load, delay, ldelay to zero;
7  Initialize β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

3.1.2 RC Delay Computation on GPU. Algorithm 2 shows the pseudocode for our GPU kernel. We optimize the data structure of RC trees in GPU memory. We launch the kernel per net under each Early/Late and Rise/Fall condition. A total of $4N$ threads are launched, where N is the number of nets. Initially, the *netID* and *condID* is computed in line 1-3. We compute the offsets of the data for the net in the arrays in line 4-5 and initialize the output arrays with zeros in line 6-7. Then, we traverse and update the values of *load* (line 8-11), *delay* (line 12-15), *ldelay* (line 16-19), *beta* and *impulse* (line 20-24).

We store the parent index of each node in array *parent*, as shown in Figure 4(b). This provides a workload-balanced parent-child representation on GPU, while preserving our ability to perform DP updates. For example, the recursive equation for *load* is

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

as shown in Figure 5(a). This equation is not directly evaluable because we cannot sum over all children of u without knowing the adjacency list. To handle this problem, we recognize each $load_u$ as a running sum. Algorithm 2 updates the running sum of u at each child of u (line



(a) Upward (b) Downward

Figure 5: Different directions of DP. In (a), the update of recursive equation goes upward. In other words, the value of a parent depends on the value of its children. In (b), the update goes downward. In other words, the value of a child depends on its parent.

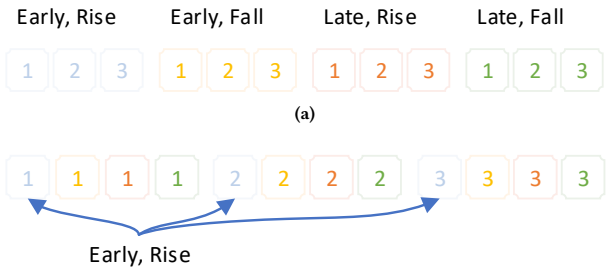


Figure 6: Memory arrangement for Early/Late and Rise/Fall cases. (a) Independent access; (b) Interleaved access.

10), progressivly leading to the final result. Because we traverse the node sequence backward (line 8-11), and every child of u appears after u, $load_u$ is ready when we encounter u in the sequence. As another example, the recursive equation for *delay* is:

$$delay_v = delay_u + pres_v \times load_v$$

where u is the parent of v, as shown in Figure 5(b). Implementation of this equation becomes straightforward using our parent index array (line 13-14). Updating *ldelay* and *beta* is similar to *load* and *delay*, respectively.

We optimize the global memory access latency when the number of nets is large and we have to update the RC at different conditions using different threads. Data stored in GPU's global memory will incur some latency when accessed. In practice, GPU will identify threads that request memory access to adjacent addresses, and coalesce these requests. Thus, we interleave the memory of the four threads performing independently on each of the four Early/Late and Rise/Fall conditions, instead of storing them separately (see Figure 6). This ensures memory requests emitted from the adjacent four threads are adjacent. The index of node i with condition index *condID* is $4i$ + *condID* (marked in Algorithm 2).

3.2 Levelization

Levelization is a preparation step for timing propagation. It builds level-by-level dependencies for propagation tasks and accounts for nearly 40% of the runtime [4] (see Figure 2). A root cause is its single-threaded pattern. Existing timers, including industrial tools [12], construct a level list data structure for all logic levels through a single-threaded

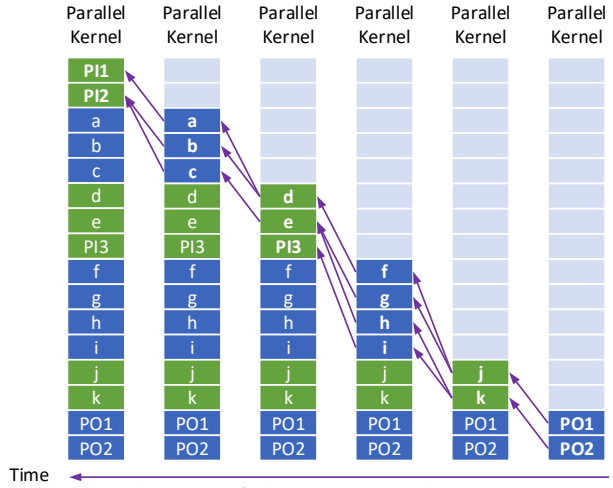


Figure 7: Levelization of the timing graph in Figure 1 using GPU. Nodes in bold are frontiers at the corresponding iteration. At each iteration, we obtain one level of nodes.

BFS or DFS. Tasks within the same level can run in parallel; Lower-level tasks must not run after tasks at higher levels. Maintaining such a data structure is time-consuming. To reduce the runtime, we develop a GPU-accelerated levelization algorithm.

Algorithm 3: Levelize

Input: the set of nodes *nodes*
Data: the adjacency list *out*, the current in-degree *in*
Output: a level list of nodes

```
1 F ← {f ∈ nodes : in_f = 0};
2 while F is not empty do
3 |   output F;
4 |   F' ← {};
5 |   Call advanceFrontier on F and get F';
6 |   F ← F';
7 end
```

The procedure of our GPU-accelerated levelization is shown as Algorithm 3. The key idea is to maintain a set of nodes in the present level, called *frontiers*, denoted as F. The initial frontiers are nodes that do not have input edges (line 1). The algorithm loops through line 3-6 until all nodes are discovered. At each iteration, we invoke a GPU kernel function `advanceFrontier` to discover the next frontiers in parallel based on the current ones.

Algorithm 4 shows the pseudocode for `advanceFrontier`. This algorithm works on current frontiers and each thread processes one frontier. We enumerate all output edges of each frontier (line 3-8). For each output edge with destination v, we decrease the in-degree of v by one. If the in-degree of v becomes zero afterwards, we add v to the set of next frontiers.

In this algorithm, we perform edge exploration of different frontiers simultaneously, while different output edges of one node are explored sequentially. However, if only a few frontiers have large output degrees (number of outgoing edges), we may encounter imbalanced workload among GPU threads. To tackle this problem, we adopt a *reverse* technique. The idea is based on the observation that in most circuit designs the input degree of a node is smaller than its output degree [9]. For example, in the million-gate design, netcard [2], the maximum input

Algorithm 4: Advance Frontier

Input: the old frontier F
Data: the adjacency list *out*, in-degree array *in*
Output: the new frontier F'

```
1  nodeID ← blockIdx.x × blockDim.x + threadIdx.x;
2  if nodeID ≥ size(F) then return;
3  for v in out[nodeID] do
4      oldvalue ← atomicAdd(in[v], -1);
5      if oldvalue = 1 then
6          Add v to F';
7      end
8  end
9  return G;
```

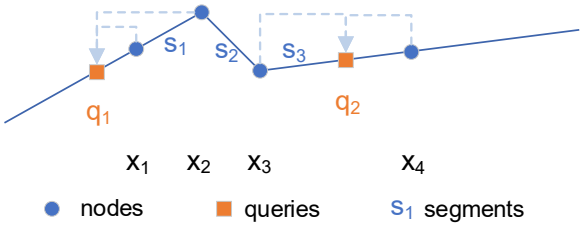


Figure 8: Example of 1D lookup table query. This 1D lookup table is essentially a piecewise linear function with three segments s_1, s_2, s_3. There are two queries q_1 (that hits s_1) and q_2 (that hits s_3). We get the result by evaluating the queried x-value on the specific segment, regardless of interpolation (like q_2) or extrapolation (like q_1).

Algorithm 5: LUT Interpolation

```
   /* Input: line (x1, y1)--(x2, y2)                          */
   /* Input: the x value queried                              */
1  Function interpolate(x1, x2, y1, y2, x):
2      if x1 = x2 then return y1;
3      else return d1 + (d2 − d1) (x−x1)/(x2−x1);
4  end
   /* Input: n × m look-up table                              */
   /* Input: the point queried (x, y)                         */
5  Function lut_lookup(n, m, X, Y, mat, x, y):
6      i' ← 0;
7      i ← min(1, n − 1);
8      while i + 1 < n and X[i] ≤ x do
9          i' ← i;
10         i ← i + 1;
11     end
12     j' ← 0;
13     j ← min(1, m − 1);
14     while j + 1 < m and Y[j] ≤ y do
15         j' ← j;
16         j ← j + 1;
17     end
18     r_i' ←interpolate(Y[j'], Y[j], mat[i', j'], mat[i', j]);
19     r_i ←interpolate(Y[j'], Y[j], mat[i, j'], mat[i, j]);
20     r ←interpolate(X[i'], X[i], r_i', r_i);
21     return r;
22 end
```

degree and output degree are 8 and 260, respectively. If we execute the levelization algorithm on the reversed timing graph (i.e., from output pins to input pins), we can acquire higher parallelism and more balanced workload during the edge exploration process. After we complete the levelization, the ordinary level orders can be retrieved by reversing these level orders. Figure 7 illustrates our levelization process. Large designs typically have hundreds of levels with enough parallelism for running thousands of independent tasks within a level.

3.3 Timing Propagation and Look-up Table

According to the runtime breakdown in Figure 2, timing propagation is fairly CPU-efficient since most LUTs are small, but we still observe modest improvement by migrating this step to GPU especially for large designs (e.g., millions of gates). The delay and slew of cell arcs are determined by the specific type of a cell. In NLDM, dealy and slew are modeled through a piecewise linear function of the input slew and the output load of the arc. Sample points are characterized in LUTs. The timer queries slew and delay through a cell arc using bilinear interpolation or extrapolation.

Algorithm 5 presents the detail of table lookup on GPU. The 2D bilinear interpolation is computed by three 1D linear interpolations (line 18-20). A 1D linear interpolation is to find the y-value of a piecewise linear function (i.e. a polyline) at a given x-value. We develop a general approach to cover both interpolation and extrapolation, as shown in Figure 8, where we only need to find which segment of linear function covers the given x-value. In the algorithm, line 6-11 finds such segment that x lies in as two consecutive indices, i' and i. We use two variables to deal with degeneralized cases of one row (or column), where $i' = i$. Since LUTs are usually small, the algorithm performs a linear search on segments for each query, and it is efficient enough on GPU.

4 EXPERIMENTAL RESULTS

We implemented our GPU-accelerated STA algorithm on top of OpenTimer [4] and evaluated the results using TAU15 contest benchmarks [2]. These benchmarks come with a golden reference generated by IBM Einstimer under the static mode and provide rigorous environment for testing and experimenting new STA algorithms. We do not compare with commercial tools (e.g., PimeTime, OpenSTA) because they do not support GPU. Also, such a comparison may not be fair because of different application scopes. We undertook all experiments on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, 1 GeForce RTX 2080 GPU, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 11.0 on a host compiler of GNU GCC-8.3.0 with C++17 standards `-std=c++17` and optimization flags `-O2` enabled. In terms of kernel execution configuration, we assigned 64 threads per block with one block for each net for Algorithm 1, 4 threads for each net (one for each Early/Late and Rise/Fall condition) with a block of 64 nets for Algorithm 2, and 128 threads per block for Algorithm 3. To overlap kernel execution with data transfers, we created three CUDA streams, one for launching kernels and the other two for transferring data between CPU and GPU. We used the state-of-the-art parallel task programming library, Cpp-Taskflow [6], that had been available in OpenTimer to describe CPU-GPU dependent tasks and let the library handle task scheduling details. We measured the runtime directly from the source of `update_timing`. This gives us precise performance evaluation in a realistic use case

of STA rather than a selected module that may result in bogus GPU speed-up numbers (e.g., 100×). All data is an average of ten runs.

4.1 Full Timing

Table 1 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 14 benchmarks, aes_core, vga_lcd, vga_lcd_iccad, b19, cordic, des_perf, edit_dist, fft, leon2, leon3mp, netcard, mgc_edit_dist, mgc_matrix_mult, and tip_master. These benchmarks were used as the final hidden testcases of TAU15 Contest to evaluate contestants' entries at a large scale. We do not include other benchmarks of small sizes (fewer than 10K gates) because their runtime results are very small (10–30 ms). We ran both OpenTimer and our algorithm using the maximum hardware concurrency of 40 CPUs and 1 GPU on our platform. Our runtime is faster than OpenTimer across all benchmarks. The three largest speed-up values we observed are 3.69× on leon2 (1.6M gates), 3.60× on netcard (1.5M gates), and 3.02× 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.

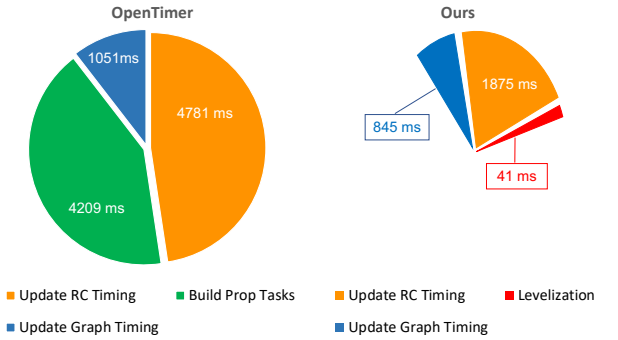


Figure 9: Runtime breakdown of the circuit leon2 (21M nodes).

Figure 9 shows the runtime breakdown of OpenTimer and our algorithm for notable items (>40 ms) on the largest circuit, leon2. OpenTimer spends 4209 ms to construct the task dependency graph for updating graph timing. Creating a large task dependency graph has non-negligible overhead because it requires additional data structures (managed by Cpp-Taskflow) to describe dependent tasks that represent the entire timing graph. Also, this process is single-threaded. 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. It takes GPU only 41 ms to accomplish the levelization using our algorithm. We observe a large amount of runtime reduction from updating RC timing. It takes 4781 ms for OpenTimer to finish RC timing whereas we reach the goal by 2.55× faster. Our runtime for updating the graph timing is a bit faster (845 ms vs 1051 ms), due to our GPU-based LUT interpolation.

Figure 10 draws the runtime scalability versus increasing numbers of CPUs on the two largest designs, leon2 and netcard. Each CPU corresponds to a user-land thread. Both methods leverage task parallelism (with Cpp-Taskflow [6]) to describe dependent tasks and update the STA graph in parallel. In our implementation, the benefit includes overlapped CPU-GPU tasks especially for data transforms and transfers. Increasing the number of CPUs can thus also speed up our algorithm. We observe both methods scale up to 10 CPUs. Performance stagnates at about 16 CPUs. Regardless of CPU numbers, our runtime is always faster than OpenTimer, and there exists a remarkable gap. The largest

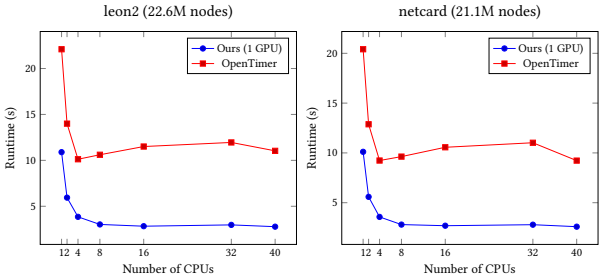


Figure 10: Runtime values at different numbers of CPUs. Our runtime under 1 CPU and 1 GPU is close to OpenTimer of 40 CPUs.

speed-up occurs at 32 CPUs, where ours is faster than OpenTimer by 4.18× on leon2. Even with a single CPU, our GPU implementation is able to finish the timing closer to that of OpenTimer using 40 CPUs (10892 ms vs 11036 ms in leon2 and 10109 ms vs 9227 ms in netcard). These results clearly demonstrate the strength of our approach.

4.2 Incremental Timing

A key reason our GPU implementation is faster than OpenTimer is the data size. Before any incremental timing takes place, the STA graph must experience at least one round of *full-timing* update, including cell delay, RC delay, slew, arrival time, required arrival time, and so on. These tasks accumulate a large amount of data and computation that can benefit from the use of GPU parallelism. During incremental timing, computation varies and may scope to a small local region or the entire timing landscape. Computational efficiency largely depends on the number of *propagation candidates* to trigger incremental timing. In OpenTimer, this is equivalent to the union of fanin and fanout cones spanned by frontier pins from which incremental timing must be issued at a minimum [4]. 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. However, we must highlight that the efficiency of incremental timing has a lot to do with the design of a STA engine and its application scope. We study the effect of important factors on the performance of our GPU algorithm.

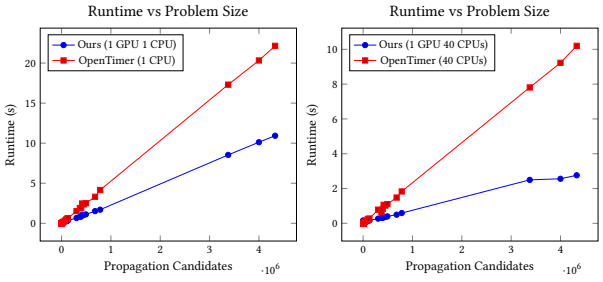


Figure 11: Runtime values at different problem sizes. Beyond about 60K propagation candidates, our runtime is always faster than OpenTimer at any CPU numbers.

Figure 11 compares of runtime at different problem sizes in terms of the number of propagation candidates reported by OpenTimer between our GPU algorithm and OpenTimer using the least and the maximum CPU concurrency. When the problem size is smaller than 10K, OpenTimer is always faster than ours. This is because launching GPU operations has certain overhead. For example, it took 1–5 ms to

Table 1: Performance comparison between OpenTimer (40 CPUs) and our GPU-accelerated Implementation (1 GPU) to complete one iteration of full timing on large circuit designs (>10K gates) of TAU 2015 contest benchmarks

Benchmark	# PIs	# POs	# Gates	# Nets	# Pins	# Nodes	# Edges	OpenTimer Runtime (40 CPUs)	Our Runtime (40 CPUs 1 GPU)	
									Runtime	Speed-up
aes_core	260	129	22938	23199	66751	413588	453508	156 ms	138 ms	1.13×
vga_lcd	85	99	139529	139635	397809	1966411	2185601	829 ms	311 ms	2.67×
vga_lcd_iccad	85	99	259067	259152	679258	3556285	3860916	1480 ms	496 ms	2.98×
b19	22	25	255278	255300	782914	4423074	4961058	1831 ms	585 ms	3.13×
cordic	34	64	45359	45393	127993	7464477	820763	274 ms	167 ms	1.64×
des_perf	234	140	138878	139112	371587	2128130	2314576	832 ms	325 ms	2.56×
edit_dist	2562	12	147650	150212	416609	2638639	2870985	1059 ms	376 ms	2.86×
fft	1026	1984	38158	39184	116139	646992	718566	241 ms	148 ms	1.63×
leon2	615	85	1616369	1616984	4328255	22600317	24639340	10200 ms	2762 ms	3.69×
leon3mp	254	79	1247725	1247979	3376832	17755954	19408705	7810 ms	2585 ms	3.02×
netcard	1836	10	1496719	1498555	3999174	21121256	23027533	9225 ms	2571 ms	3.60×
mgc_edit_dist	2562	12	161692	164254	450354	2436927	2674934	1021 ms	368 ms	2.77×
mgc_matrix_mult	3202	1600	171282	174484	492568	2713241	2994343	1138 ms	377 ms	3.02×
tip_master	778	857	37715	38493	95524	533690	570154	163 ms	143 ms	1.14×

 # 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

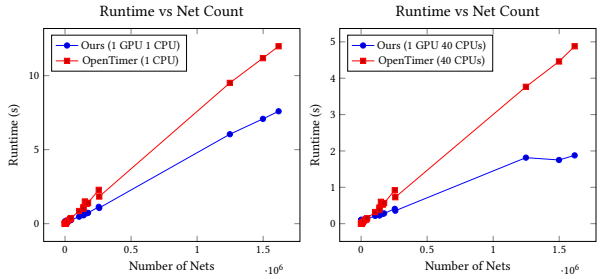


Figure 12: Runtime values at different net counts. Beyond about 40K nets, our GPU-accelerated RC computation is always faster than OpenTimer, regardless of CPU numbers.

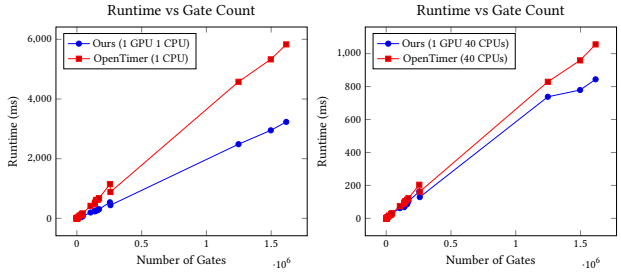


Figure 13: Runtime values at different number of gates (~LUT numbers). Beyond about 45K gates, our GPU-accelerated LUT interpolation becomes faster than OpenTimer.

allocate large global memory in our GPU. Yet the runtime difference between our algorithm and OpenTimer on small designs 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. We observe consistent trend regardless of the number of CPUs used to run both methods. Figure 12 plots the runtime growth of updating RC timing with increasing number of nets. Each net has about 10–100 internal nodes in the RC tree. The runtime growth resembles that of Figure 11 because updating RC timing occupies the majority of runtime. At small net count (fewer than 40K nets), we observe little benefit of GPU due to the data and kernel overheads. After that, our GPU-accelerated RC timing computation is consistently faster than OpenTimer at any CPU numbers. For example, at 45K nets, we are 1.16× and 1.52× faster than OpenTimer under 1 and 40 CPUs. Figure 13 draws the runtime growth of updating graph timing with increasing number of gates. The result approximates the impact of LUT counts on the runtime. At about 45K gates (roughly 360K LUTs), ours GPU-accelerated LUT interpolation algorithm starts outperforming OpenTimer. It is expected as we increase the number of CPUs, the performance margin between both methods become close as LUT interpolation is less data- and compute-intensive than other tasks. 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 [21–23].

5 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 a STA graph. We have leveraged task parallelism to describe dependent CPU-GPU tasks such that data processing and kernel computation are efficiently overlapped. Compared to the state-of-the-art STA engine, OpenTimer, we achieved up to 3.69× speed-up on a large design of 1.6M gates and 1.6M nets using 1 GPU. Our future work includes developing GPU-accelerated algorithms for different delay calculators and incorporating GPU task parallelism using CUDA graph feature to reduce the overhead of CUDA streams. We also plan to accelerate path-based timing analysis [24, 25] using GPU and integrate our timer into [26, 27].

ACKNOWLEDGE

This project is supported in part by the Beijing Municipal Science and Technology Program (No. Z201100004220007) and the National Key Research and Development Program of China (No. 2019YFB2205000).

REFERENCES

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

[2] 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.

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

[4] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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 *Proceedings of the 2014 on International Symposium on physical design*, 2014, pp. 153–160.

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

[15] 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.

[16] 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.

[17] H. H.-W. Wang, L. Y.-Z. Lin, R. H.-M. Huang, and C. H.-P. Wen, "Casta: Cuda-accelerated static timing analysis for VLSI designs," in *2014 43rd International Conference on Parallel Processing*. IEEE, 2014, pp. 192–200.

[18] 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.

[19] 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.

[20] J. Cong, K. Gururaj, W. Jiang, B. Liu, K. Minkovich, B. Yuan, and Y. Zou, "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.

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

[22] N. Viswanathan, G.-J. Nam, J. A. Roy, Z. Li, C. J. Alpert, S. Ramji, and C. Chu, "ITOP: Integrating timing optimization within placement," in *ISPD*, 2010, pp. 83–90.

[23] D. Liu, B. Yu, S. Chowdhury, and D. Z. Pan, "TILA-S: Timing-driven incremental layer assignment avoiding slew violations," *IEEE TCAD*, 2017.

[24] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A general cache framework for efficient generation of timing critical paths," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019.

[25] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "A general cache framework for efficient generation of timing critical paths," in *Proceedings of the 57th Annual Design Automation Conference 2020*, 2020.

[26] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," *IEEE TCAD*, 2020.

[27] Y. Lin, W. Li, J. Gu, H. Ren, B. Khailany, and D. Z. Pan, "ABCDPlace: Accelerated batch-based concurrent detailed placement on multi-threaded cpus and gpus," *IEEE TCAD*, 2020.