A GPU-Accelerated Framework for Path-Based Timing Analysis

Guannan Guo, Tsung-Wei Huang, *Member, IEEE*, Yibo Lin, *Member, IEEE*, Zizheng Guo, Sushma Yellapragada, and Martin D. F. Wong, *Fellow, IEEE*

***Abstract*—As a key routine in static timing analysis (STA), path-based analysis (PBA) plays a very important role in refining the critical path report by reducing excessive slack pessimism. PBA is also well known for its long execution time, which makes it a hot topic for parallel computing in the STA community. However, nearly all of the parallel PBA algorithms are restricted to CPU architectures, which greatly limits their scalability. To achieve a new performance milestone on PBA, we must leverage the high throughput computing in the graphics processing unit (GPU). Therefore, in this work, we propose a new GPU-accelerated PBA framework which contains compact data structures and highly efficient kernels. By integrating with GPU-accelerated preprocessing steps, our framework can also effectively handle extensive critical path constraints. Besides, we highlight many optimization techniques that can overcome the execution bottleneck and further boost the performance. In experiments, we demonstrate 543× speed-up compared to the state-of-the-art PBA algorithm on the design with 1.6 million gates, which outperforms 25×–45× over the state-of-the-art parallel PBA algorithm on 40 CPU cores. A fully optimized framework can achieve 3×–5× speed-up on top of that.**

***Index Terms*—Graphics processing unit (GPU) acceleration, static timing analysis (STA).**

I. Introduction

PATH-BASED analysis (PBA) is a common practice to remove path slack pessimism and obtain accurate timing report in the static timing analysis (STA) tool [1]. Besides the high accuracy, PBA is well-known for its high runtime complexity, typically 10-1000× greater than the complexity of graph-based analysis (GBA) [2]. Due to this high complexity, PBA is very daunting to designers at the early stage of the design closure flow, even though designers appreciate PBA's accuracy benefit.

Manuscript received 3 May 2022; revised 23 August 2022; accepted 3 October 2022. Date of publication 9 May 2023; date of current version 20 October 2023. This work was supported by the National Science Foundation (NSF) under Grant CCF-2126672, Grant CCF-2144523 (CAREER), and Grant OAC-2209957. This article was recommended by Associate Editor X. Li. *(Corresponding author: Guannan Guo.)*

Guannan Guo and Sushma Yellapragada are with the Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Champaign, IL 61821 USA (e-mail: gguo4@illinois.edu).

Tsung-Wei Huang is with the Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84124 USA.

Yibo Lin and Zizheng Guo are with the School of Integrated Circuits, Peking University, Beijing 100871, China.

Martin D. F. Wong is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.

Digital Object Identifier 10.1109/TCAD.2023.3272274

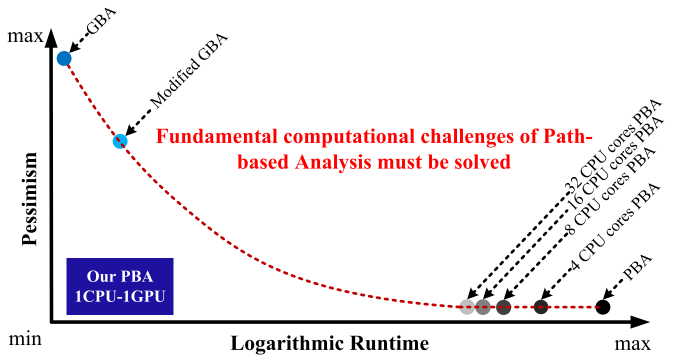


Fig. 1. Computational tradeoff between runtime and pessimism reduction on path-based timing analysis.

We use Fig. 1 to illustrate the progress made by the timing literature and we want to highlight that the computational challenges of PBA remain unsolved with CPU parallelism. Current multicore CPU parallelism cannot provide groundbreaking performance improvement to PBA in the STA engines. A transformational performance milestone can only be achieved with the power of heterogeneous parallelism or CPU-graphics processing unit (GPU) hybrid computing. Nevertheless, it is extremely challenging to offload the PBA process on GPU for the following three reasons. First, PBA is graph-oriented and contains highly irregular computational patterns. We need to identify and dispatch the computation-intensive workload to CPU and throughput-intensive workload to GPU. Second, path generation process is highly dynamic, which requires specially designed GPU kernels to generate critical paths and maintain relative path priorities. Finally, to generate million of paths on million-gate designs, we need efficient and compact data structures to overcome the hurdle of limited GPU memory.

In this work, we propose a novel framework that successfully resolves these challenges and accelerates PBA with transformational speed. We focus on parallelizing the core procedure, *critical path generation*, which takes the majority of the PBA execution time [1]. Specifically, in this procedure, we identify a set of critical paths from an updated STA graph and rank them in decreasing slack values. PBA algorithms or frameworks can then perform path-specific update on this path set. Our key contributions are listed as follows.

1) *GPU-Accelerated Path Search Algorithm:* We propose a new iterative path search algorithm that maximize the GPU computation throughput in each iteration. Before

the iteration begins, we leverage thousands of GPU threads to construct a forest that roots at circuit datapath endpoints and unifies all path suffix informations. In the main path search iteration, different GPU threads are dispatched to alternate different critical path prefixes from the previous iteration.

2) *GPU-Efficient Data Structures:* To efficiently utilize the GPU memory, we maintain all explored critical paths in a dense array structure. We leverage the implicit path representation that saves each critical path in constant memory complexity. Besides, we regularly verify the number of explored path candidates and remove path candidates with low priority. In this way, we also improve the overall memory usage during the entire path search process.
3) *Scalable to Large Numbers of Critical Paths:* We organize the search process such that there are no shared memory conflicts between different GPU threads. The path exploration workload is determined by the path prefix. By dispatching GPU threads to different combination of path prefixes, our critical path generation process can scale to millions of paths.
4) *Extensive Path Constraint Handling:* Our path search algorithm can efficiently support filter kernels to remove unwanted paths that do not satisfy the given path constraints. We construct a special ranking array and a scanning array to label regions in the constrained subgraph. These two arrays act as efficient filters and maintain high data throughput for our iterative path exploration algorithm.

An industrial standard timer [3] provides us the golden reference on real circuit designs to run our evaluations. We verify our timing report matches the golden reference. We use state-of-the-art path generation algorithm [4], [5] as baseline. In our experiments, we achieve up to 543× speed-up over the baseline. At the extreme case, we achieve 25×–45× faster than the baseline running on 40 CPU cores. We also prove our framework can handle extensive path constraints. We randomly select common pin sequences in full timing report and choose them as path constraints. Our algorithm can generate critical paths that fully satisfy the path constraints. Additionally, we provide several fine-grained optimizations that can further improve the performance of our framework. We believe our framework can help designers to incorporate PBA earlier in the design flow. By adopting our framework, STA engines could improve quality of results (QoR) with reasonable turnaround time.

II. Path-Based Timing Analysis

PBA plays a very pivotal role in STA [1]. Its goal is to remove excessive slack pessimism introduced from GBA. In STA, GBA is usually performed first to propagate the timing information across the STA graph. These timing information include slew, delay, required arrival time, and arrival time. Since these timing values are propagated under the worst-case scenario (i.e., early or late mode) [1], identified critical paths often have very pessimistic slack values. Therefore,

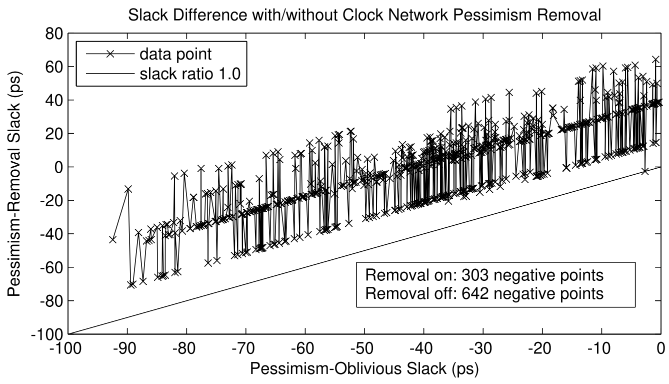


Fig. 2. Example of pessimism reduction on critical paths [4].

STA requires PBA to make path-specific updates and remove unwanted pessimism. For example, common path pessimism removal (CPPR) and advanced on-chip variation (AOCV) are both common PBA workloads to achieve this objective [4]. Fig. 2 illustrates an example that PBA can reduce the absolute slack values by half with pessimism reduction on critical paths reported by GBA. For all the PBA workloads, one core routine is to generate a set of critical paths for reanalysis. This routine is very time consuming and may take hours to complete because the STA engine needs to search for critical paths from an exponential number of path candidates. Thus, the designers have emphasized that EDA vendors should seek out new parallel paradigms to boost the runtime performance of the PBA workloads [6].

III. Related Works

To alleviate the time-consuming PBA workload, there are various existing works [4], [5], [7], [8], [9], [10], [11], [12], [13] that seek improvement. However, their improvements are rather limited and all of them stay in the scope of CPU parallelism. For example, the state-of-the-art PBA algorithm [4], [7] proposes task-based parallelism to address the workload dependency in the STA graph, but its performance improvement stagnates at 16 cores. The PBA workload becomes more dynamic and complicated if the path constraints are considered. Works [14] and [15] improve PBA considering path constraints on CPU architectures, but their algorithms are mostly sequential and lack parallelization benefits. Recently, Guo et al. [16] proposed a new GPU-accelerated PBA algorithm that leverages GPU's high throughput computation and boosts PBA to a new performance milestone. Additionally, Guo et al. [17] also proposed GPU-friendly preprocessing kernels to address the path constraint handling. Both works greatly accelerate the process to explore critical paths, because exploring an exponential number of paths can be highly throughput driven. GPU is built for throughput-driven applications. It has many lightweight threads to compute data vectors at one time. Multiple GPU threads can work together to compute a block of data in a single or the same instruction. However, since both works choose to mange the critical path priority on the CPU, this workload becomes the major bottleneck of the entire PBA application. To overcome

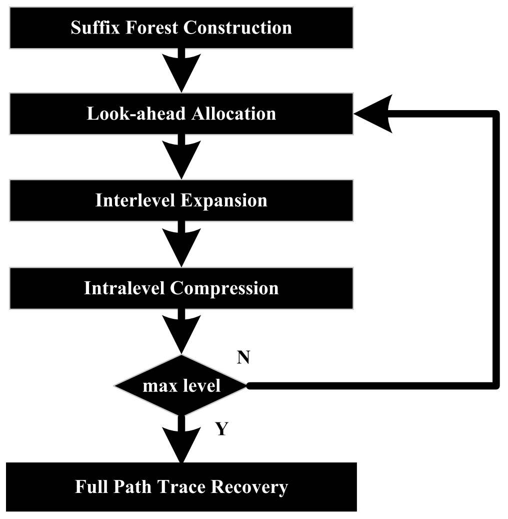


Fig. 3. Overview of core GPU-accelerated PBA algorithm.

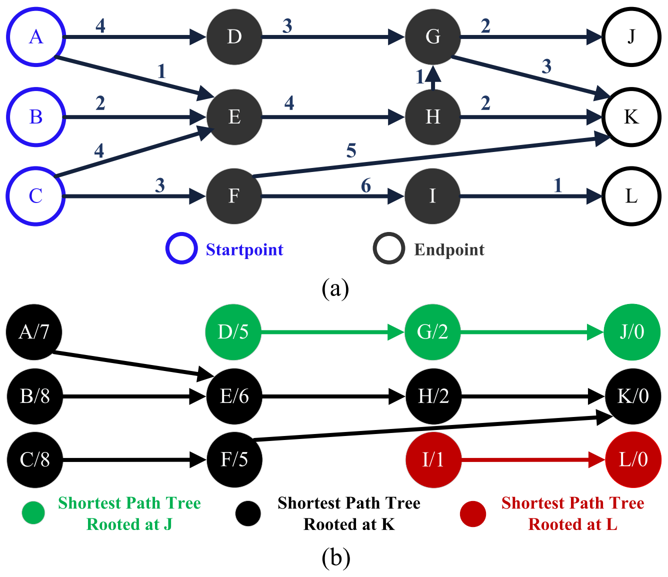

A
B
C
D
E
F
G
H
I
J
K
L
4
1
2
4
3
3
4
5
6
2
1
3
2
1
Startpoint
Endpoint
(a)
A/7
B/8
C/8
D/5
E/6
F/5
G/2
H/2
I/1
J/0
K/0
L/0
Shortest Path Tree Rooted at J
Shortest Path Tree Rooted at K
Shortest Path Tree Rooted at L
(b)

Fig. 4. Shortest path forest generation on GPU. (a) STA graph. (b) Shortest path forest.

this bottleneck, we propose a new framework that maintains the high path exploration throughput and manage priorities completely on GPU. Compared to the previous works, our framework completes the entire path generation process on GPU that greatly reduces the execution and communication cost on CPU.

IV. Proposed GPU-Accelerated PBA

The flow diagram of our core GPU-accelerated PBA algorithm is shown in Fig. 3. We use an iterative approach to explore the critical path candidates. Before the main iteration begins, we unify all the path suffix information of an updated STA graph in a suffix forest. The suffix forest is essentially a shortest path forest where its roots are flip-flop inputs and primary outputs. In the main iteration, we explore new paths by alternating path prefixes with edges not belonging to the suffix forest. We denote these edges as *deviation edges*. We organize critical paths in levels which is equal to the number of *deviation edges* in the path. We increment the level for each iteration. One iteration is composed of three steps: 1) look-ahead level allocation; 2) interlevel expansion; and 3) intralevel compression. Details of each step can be referred to later sections. We can stop the iteration either when we reach the maximum level or we have sufficient critical paths for an accurate report. In the end, we will perform path recovery to generate full path trace from our implicit representation.

A. STA Graph Structure on GPU

In most STA engines, the circuit graph is modeled as a directed acyclic graph (DAG). A pin and transition combination is modeled as a vertex. A pin-to-pin connection is modeled as an edge. To make the STA graph as compact as possible, we choose the compressed sparse row (CSR) graph representation. CSR is commonly used as a condensed graph format in GPU applications [18]. It contains three linear arrays which hold information for vertex offsets, edge destinations, and edge weights, respectively. Given a directed graph with N vertices and M edges, its CSR format requires $N + 2M$ memory. Since we need to store both the fan-in STA graph G^- and fan-out STA graph G^+ in CSR, the total memory requirement for two directed STA graphs is $2N + 4M$.

B. Suffix Forest Construction

Before we alternate path prefix for each critical path, we hold all the path suffix information in a compact data structure, *suffix forest* or *shortest path forest*. The path prefix-suffix representation has been explored in previous works [4], [5]. But instead of summarizing path suffix from all datapath endpoints, an independent shortest path tree at each endpoint is constructed regardless of overlapped regions between trees. These overlapped regions will lead to redundant tree construction, which implies high runtime and memory cost. Instead, we unify all shortest path trees into a forest that includes a predecessor array `forest[N]` and a distance array `distances[N]`. Our forest does not duplicate any vertex, which eliminates redundant construction at the same vertex. We illustrate our *shortest path forest* [Fig. 4(b)] in Fig. 4 given the STA graph [Fig. 4(a)]. We can observe three shortest path trees rooted at vertices J, K, and L. Their union forms the shortest path forest. We do not fully grow trees rooted at J and L to the start, because we have the tree K with smaller cumulative distances at the overlapped vertices. We can stop growing tree from root J at vertex D and stop growing tree from root L at vertex I. With this simple example, we can already demonstrate that our *shortest path forest* is much more compact and computation-efficient, because we save the effort to traverse the shared logic for different endpoints.

Moreover, we can utilize high GPU throughput to construct our *shortest path forest*. Our GPU kernels are inspired by the GPU-accelerated single-source shortest path algorithm [19], [20]. In our kernel, all datapath endpoints are marked as roots or destinations. We initialize these destinations with

Algorithm 1: Distance Propagation Kernel

```
Input  : fan-in STA graph $G^-$ in CSR with $N$ vertices and
         $M$ edges: vertices[$N$], edges[$M$], weights[$M$]
Input  : Distance buffer, distanceBuffer[$N$]
Input  : Bitmap for recently updated vertices, bitmap[$N$]
Result: Distances array, distances[$N$]
1  threadid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
2  if threadid ≥ $N$ then
3      return;
4  end
5  if distanceBuffer[threadid] is false then
6      return;
7  end
8  bitmap[threadid] ← false;
9  edgeFront ← vertices[threadid];
10 edgeBack ← (threadid == N-1) ? M : vertices[threadid+1];
11 for edgeid ← edgeFront to edgeBack do
12     n ← edges[edgeid];
13     weight ← weights[edgeid];
14     newD ← distances[threadid] + weight;
15     atomicMin (&distanceBuffer[n], newD);
16 end
17 return;
```

required arrival time. Then we propagate the shortest distances in parallel until distances converge. Algorithm 1 shows one step of concurrent propagation. We first assigns each thread a vertex (lines 1 and 2). If the assigned vertex has received a recent update (line 5), we propagate new distances to adjacent neighbors (line 15). We finalize the distance array `distances[N]` if no entry in the array can be relaxed. After all shortest distances converge by running Algorithm 1, we launch another kernel that traces back edges which contribute to the shortest distance array. We recover these edges in the predecessor array `forest[N]`. In this way, we can quickly construct the *shortest path forest* because the GPU kernel updates thousands of vertices concurrently at each iteration and we launch another one-time kernel to recover the predecessor edges.

C. Look-Ahead Level Allocation

Before exploring new path candidates, sufficient memory should be allocated beforehand. Besides, the address of the children paths can be assigned to avoid memory collision in the later expansion step. Therefore, in this step, we compute the children path number that each parent path can expand to and perform a prefix sum computation. The result of the prefix sum can be used as row offsets for memory writing, because each path occupies a fixed size of memory. In our framework, each path is implicitly represented by one *deviation edge*. The *deviation edges* are fan-out STA graph edges not contained in our suffix forest. Since there can be a sequence of *deviation edges* in a path, we only use the last one to represent the full path. The path represented by the second last *deviation edge* is the parent critical path. We organize critical paths by counting the number of *deviation edges* in the path. This number is denoted as `level`. Paths with the same `level` are maintained together in a compact linear array. Therefore, we can use data fields in Table I to fully represent a critical path.

TABLE I
Deviation Edge DATA FIELDS

Field	Definition
`level`	number of deviations edges in the path
`from`	deviation source vertex
`to`	deviation destination vertex
`parent`	parent edge in the previous level
`childOffset`	row offset of children in next level
`slack`	critical path slack

Algorithm 2: Children Path Number Computation Kernel

```
Input  : Fan-out STA graph $G^+$ in CSR with $N$ vertices and
         $M$ edges, vertices[$N$], edges[$M$]
Input  : Suffix forest, forest[$N$]
Input  : thisLevel as current level of deviation edges
Input  : levelLength as the number of deviation edges in
         thisLevel
Result: Compute path numbers originated from current level
1  threadid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
2  if threadid ≥ levelLength then
3      return;
4  end
5  v ← thisLevel[threadid].to;
6  /* Get deviation paths count originating
      from vertex v in the suffix forest   */
7  pathCount ← getPathCount(v, vertices, edges, forest);
8  thisLevel[threadid].childOffset ← pathCount;
9  return;
```

Besides `level` and `parent` that organize our data structures, we use `from` and `to` to keep track of the deviation location, `childOffset` to save the memory location of the children paths, and `slack` for the path slack value.

Algorithm 2 shows the kernel to compute path count in next level to prepare for expansion. We first assign each parent *deviation edge* in current level with one GPU thread (lines 1 and 5). We use each thread to compute the number of child deviations (line 7) and save the number in `childOffset` (line 8) data field. We then launch prefix-sum kernels to obtain the correct offset. After the prefix scan, we can find the size of next level in the last offset value. This size of next level can be used for dynamic allocation. All the other offset values represent the memory locations of the children *deviation edge*. After prefix scan, we make sure there are zero memory access conflicts between different parent paths.

D. Interlevel Expansion

In this step, we expand children critical paths by completing their corresponding *deviation edge* data fields. An example of this process is shown in Fig. 5, where we expand our STA graph in Fig. 4(a) from level 0 to level 1. We use level 0 to denote the set of shortest paths, $\{P_{\text{AEHK}}, P_{\text{BEHK}}, P_{\text{CFK}}\}$. An explicit representation for this path set is $\{e_{\emptyset A}, e_{\emptyset B}, e_{\emptyset C}\}$, where we use a leading empty set to represent a virtual edge connecting to a datapath startpoint.

When we expand a parent path to its children, we pick up the last vertex deviated by the parent and walk along the suffix forest. We scan all the *deviation edges* in this traversal.

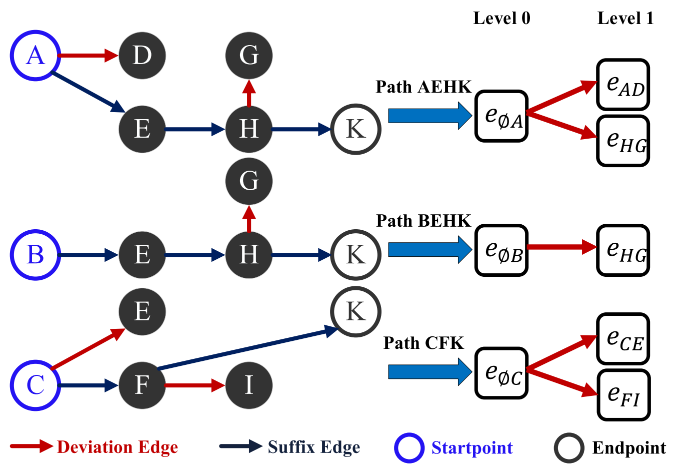


Fig. 5. GPU expansion of the first level.

These *deviation edges* implicitly represent the children paths. For example, if we expand parent path $e_{\emptyset B}$ or P_{BEHK}, we pick up from vertex B and walk through BEHK. Along this way, we encounter one *deviation edge* e_{HG}. As shown in Fig. 5, path e_{HG} or P_{BEHGJ} is the only child path of parent $e_{\emptyset B}$. This expansion can run concurrently with other parent paths in the same level. For each newly expanded child path e_{vu}, the slack of child s_{child} can be computed given the parent's slack s_{parent}

$$s_{\text{child}} = s_{\text{parent}} + \text{distance}(u) + \text{weight}(e_{vu}) - \text{distance}(v).$$

As shown in Algorithm 3, we integrate the expansion rules above in our new level expansion kernel. We launch the kernel and assign each parent *deviation edge* to a single GPU thread (line 1). We first locate the child path starting memory location in the next level by using `childOffset` (lines 5, 17, and 24). Then we pick up from the last deviated vertex of the parent (line 8) and walk along the suffix forest (line 28). We collect all the *deviation edges* (line 15) along the way and complete the information for the children paths (line 17–23). Our kernel can maximize the throughput for children path exploration, because the memory writing locations are distinctly separated for different parent paths. Therefore, no shared memory synchronization is needed in our expansion kernel.

E. Intralevel Compression

In each iteration, we expand a new level of critical paths based on previous critical paths. If we cascade these expansions without any restriction, we will quickly exhaust the GPU memory in a few iterations, because redundant paths in the earlier level will certainly generate more redundant children paths in the later expansion. To address the high memory consumption, we remove redundant critical paths right after each expansion so that they can no longer generate more children paths. We can prove the path removal does not impact the QoRs. Assume a path in the previous level should be removed, but some descendant path in subsequent levels should be included in the critical paths. The slack values from parent to child paths are nondescending, because the child path has one more deviation edge and each deviation edge adds a nonnegative delay value to the parent slack. Therefore, if some

Algorithm 3: New Level Expansion Kernel

```
   Input  : Fan-out STA graph $G^+$ in CSR with $N$ vertices and
            $M$ edges, vertices[$N$], edges[$M$], weights[$M$]
   Input  : Suffix forest, forest[$N$] as edge array, distances[$N$]
            as distance array
   Input  : thisLevel as current level of deviation edges
   Input  : nextLevel as next level of deviation edges
   Input  : levelLength as the length of current level
   Result: Explore children critical path candidates in next level
 1 threadid ← blockIdx.x * blockDim.x + threadIdx.x;
 2 if threadid ≥ levelLength then
 3     return;
 4 end
 5 offset ← (threadid == 0) ? 0:
     thisLevel[threadid-1].childOffset;
 6 level ← thisLevel[threadid].level;
 7 slack ← thisLevel[threadid].slack;
 8 v ← thisLevel[threadid].to;
 9 while v is not endpoint do
10     edgeFront ← vertices[v];
11     edgeBack ← (v == N-1) ? M : vertices[v+1];
12     for edgeid ← edgeFront to edgeBack do
13         n ← edges[edgeid];
14         weight ← weights[edgeid];
15         if edgeid is deviation edge then
16             /* Fill out child deviation edge
                  data fields                    */
17             childPath ← nextLevel[offset];
18             childPath.level ← level+1;
19             childPath.from ← v;
20             childPath.to ← n;
21             childPath.parent ← threadid;
22             childPath.childOffset ← 0;
23             childPath.slack ← slack + distances[n] +
                 weight - distances[v];
24             offset ← offset + 1;
25         end
26     end
27     /* Traverse along the suffix forest   */
28     v = forest[v];
29 end
30 return;
```

descendant path in subsequent levels should be included in the critical paths, then all its predecessors should be included, which contradicts the assumption. Given an STA graph with vertex number N, edge number M, graph diameter d, and maximum fanout number $f_{\text{out}}^{\max}$, we expand each new level in $O(df_{\text{out}}^{\max})$ times larger size. Immediately after each expansion, we sort the newly expanded level based on path slacks and keep only top k candidates, where k represents the final number of critical paths in the report. Since we can expand no more than diameter d iterations, the worst total memory complexity is $O(N+M+dkf_{\text{out}}^{\max})$. CSR graph format and final path report requires static memory $O(N+M+dk)$. Dynamic allocation of new level requires $O(dkf_{\text{out}}^{\max})$ memory during runtime.

F. Critical Paths Recovery

After the main iteration completes, we need to recover the full path trace from the implicit *deviation edge* representation. The number of iterations to expand is denoted as maximum deviation level (MDL), we make MDL a tunable parameter to users based on different accuracy and runtime needs. To ensure no loss of accuracy, users can always set MDL as the graph diameter. As discussed in Section IV-E, slack values from path to child paths are nondescending. Even with low MDL value, we also ensure that paths with higher criticality show up in the report. To obtain the final report, we merge sorted arrays from all levels and pick k *deviation edges* with worst slack values. For each *deviation edge* in the final report, we backtrace `parent` until we reach the root. We collect the list of prefix edges during backtracing. Then we complement these prefix edges with suffix edges in the shortest path forest to recover the explicit path trace. For example, if we select $e_{\emptyset C} \rightarrow e_{\mathrm{CE}}$ in Fig. 5 in our report, we recover the full path trace with forest edges in Fig. 4(b) and recover the full critical path *CEHK*. Given STA graph diameter d and report path number k, critical path recovery only requires $O(kd)$ time complexity, which can run fast enough on CPU. We can also use GPU to accelerate this process by assigning one thread to recover one critical path.

V. Extensive Path Constraints Handling

PBA with path constraints is very common in the STA tool, because optimization flows often invoke PBA repeatedly with different path constraints to verify correct timing behavior under certain logic cone. However, the path constraints make the search process more irregular and dynamic, which is harder to parallelize. In this section, we explain and demonstrate how our GPU-accelerated PBA framework can consider extensive path constraints by adding some key preprocessing steps. Our framework can support multiple main path constraints, which are listed as follows.

1) *-max_path:* The maximum number of critical paths in the timing report, which is often denoted as k. Optimization flows can control the number of paths to examine by setting this number. In this work, we experiment on a larger number of critical paths, compared to previous works that only test on $k < 32$ [14], [15].
2) *-from:* This argument fixes the datapath starting pin or location. The argument itself can be optional. We can also pair it with an optional transition (rise or fall).
3) *-through:* Each occurrence of this argument specifies a pin that the critical path must pass through. If a list of this argument is given, then all reported critical paths must contain the list of pins in the same order.
4) *-to:* This argument fixes the datapath ending pin or location. The argument itself can be optional. We can also pair it with an optional transition.

Constraints listed above are the most common and important ones which are used in the STA tools. We also support other constraint type like `-split`, which defines the corner (min or max). We enable our GPU-accelerated PBA framework to support these extensive constraints effectively by adding key preprocessing steps that are scalable on GPU. An overview of these preprocessing steps are shown in Fig. 6. The first preprocessing step is global ranking, where we assign each vertex with a rank value based on the vertex's connectivity. Then we use the rank values and sequence in path constraints

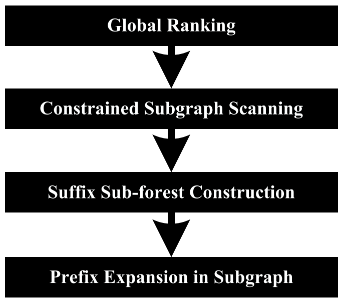

Global Ranking
Constrained Subgraph Scanning
Suffix Sub-forest Construction
Prefix Expansion in Subgraph

Fig. 6. Overview of preprocessing steps for path constraints handling.

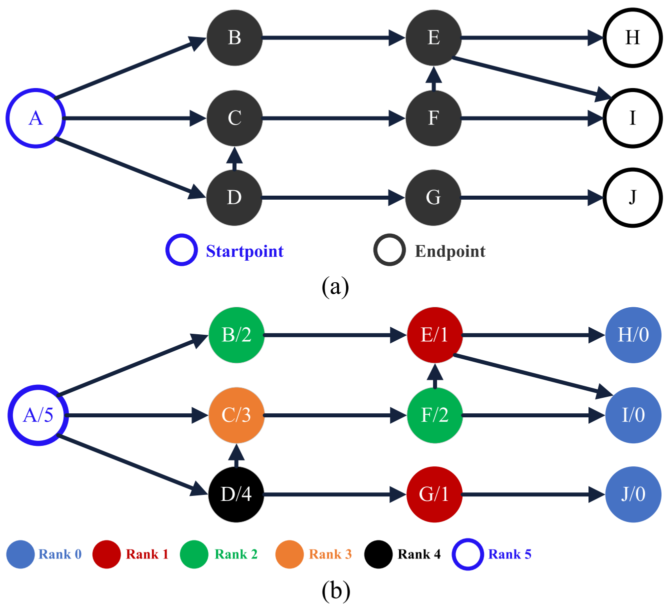


Fig. 7. Global ranking on GPU. (a) STA graph example. (b) Global ranking results.

to scan the STA graph. Based on scanned results, we label the constrained subgraph and perform path generation process under some filtering rules.

A. GPU-Accelerated Ranking

We implement a global ranking strategy to save the connectivity information of the STA graph. We denote the rank value as the maximum number of edges to reach one of the datapath endpoints. We allow multiple vertices to share the same rank values. Our ranking strategy preserves the property of topological orders. For any two vertices of an edge $e_{u \rightarrow v} \in E$, their rank values follow the property $\mathrm{rank}(u) > \mathrm{rank}(v)$.

Given an STA graph in Fig. 7(a), we rank its vertices and obtain results in Fig. 7(b). We initialize all endpoints, H, I, and J, with rank value 0. Then we let rank values propagate backwards. As an example to this backward propagation, given vertex F adjacent to the endpoint I, we assign the rank value as $\mathrm{rank}(F) = 2$, because the longest path to reach one of the endpoints is $F \rightarrow E \rightarrow H$ instead of $F \rightarrow I$.

Algorithm 4 shows the rank propagation kernel on GPU. We launch a 2-D kernel where we assign each thread in the x-axis as a pin (line 1) and each thread in the y-axis as a transition. We choose this thread assignment because two transitions of

Algorithm 4: Rank Propagation Kernel

```
Input  : fan-in STA graph $G^{-}$ in CSR with $N$ vertices $M$ as
         edges: vertices[$N$], edges[$M$], weights[$M$]
Input  : Previous rank values, ranks[$N/2$]
Input  : Rank buffer, rankBuffer[$N/2$]
Input  : Bitmap for recently updated vertices,
         rankUpdated[$N/2$]
Result: Rank values array, ranks[$N/2$]
1  pinId ← **blockIdx**.x ∗ **blockDim**.x + **threadIdx**.x;
2  riseFall ← **threadIdx**.y;
3  threadid ← 2*pinId + riseFall;
4  **if** *threadid* ≥ $N$ **then**
5      **return**;
6  **end**
7  **if** *rankUpdated[pinId]* **is false then**
8      **return**;
9  **end**
10 rankUpdated[pinId] ← **false**;
11 edgeFront ← vertices[threadid];
12 edgeBack ← (threadid == N-1) ? M : vertices[threadid+1];
13 **for** *edgeid ← edgeFront to edgeBack* **do**
14     neighborPin ← edges[edgeid]/2;
15     newRank ← ranks[pinId] + 1;
16     **atomicMax** (&rankCache[neighborPin], newRank);
17 **end**
18 **return**;
```

the same pin are stored in adjacent memory locations (line 3), which can maximize memory coalescing. For all vertices with recent rank updates (line 7), we propagate their rank values plus one to their neighbors (lines 15 and 16). All the neighbors who receive new rank values will validate the update taking the maximum over its previous rank value. We increase the GPU throughput by allowing different threads writing into multiple memory locations. To ensure correctness, we enforce synchronization by using the atomic operations (line 16). In this way, atomic operations are distributed over different memory locations. This kernel introduces less thread contention compared to topological sort which relies on a single shared counter to distribute the unique value.

B. Constrained Subgraph Scanning

We reuse the ranking results to mark out the constrained subgraph. We mark vertices with different labels to differentiate the progress of meeting the path constraints. For high memory efficiency, we use a 1-D array `labels[N]` for all the label information. We define the following relationship between label and rank values. For each adjacent pins u, v in the constraints, the label value of vertex matching pin v is the rank of pin u. For instance, if `-through u -fall_through v` is our path constraints, we define the following rule `labels[2*v+1] = rank[u]`. For the first pin in the sequence of path constraints, we use the maximum integer as its label.

Given the STA graph in Fig. 7(a) and path constraints `-through C -through E`, we obtain the scanning results in Fig. 8. For simplicity, we consider one pin as one vertex in the graph. We initialize all labels as zeros first. All the fanout pins of E (not including) get labels of the rank of last `through` pin. As shown in Fig. 8, we assign vertices H and I

Algorithm 5: Subgraph Scanning Kernel

```
Input  : Fan-in STA graph $G^{-}$ in CSR format with $N$
         vertices and $M$ edges, vertices[$N$], edges[$M$],
         weights[$M$]
Input  : Rank array, ranks[$N/2$]
Input  : Label array, labels[$N$]
Input  : Label buffer, labelBuffer[$N$]
Input  : Bitmap indicating vertices with updated labels,
         labelBitmap[$N$]
Result: Finalized label array, labels[$N$]
1  threadid ← **blockIdx**.x ∗ **blockDim**.x + **threadIdx**.x;
2  pinid ← threadid/2;
3  **if** *threadid* ≥ $N$ **then**
4      **return**;
5  **end**
6  **if** *labelBitmap[threadid]* **is false then**
7      **return**;
8  **end**
9  prevRank ← labels[threadid];
10 labelBitmap[pinid] ← **false**;
11 edgeFront ← vertices[threadid];
12 edgeBack ← (threadid == N-1) ? M : vertices[threadid+1];
13 **for** *edgeid ← edgeFront to edgeBack* **do**
14     n ← edges[edgeid];
15     neighborPin ← n/2;
16     **if** *ranks[neighborPin] < prevRank* **then**
17         labelBuffer[n] ← prevRank;
18     **end**
19 **end**
20 **return**;
```

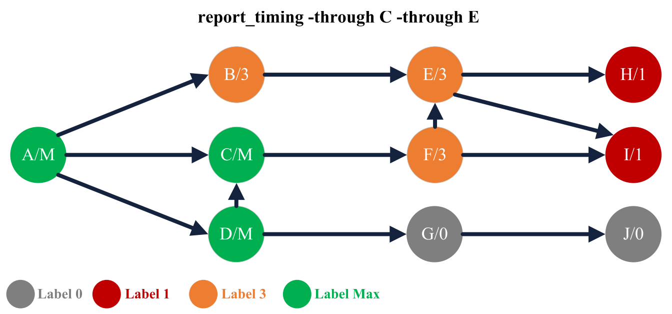


Fig. 8. Subgraph scanning results.

with labels 1. Similarly, we assign vertices from E up to C (not including) with labels as rank value of C. All vertices C and upward get labels `INT_MAX`. Since vertices G and J are not contained in the constrained subgraph, their labels remain as 0.

Algorithm 5 outlines our scanning kernel. We propagate labels to the fan-in neighbors until the rank of the fan-in neighbor is less or equal to the propagated label. We assign each thread to a vertex in the STA graph (line 1). For the recently updated vertex (line 5), we propagate the label value to its fan-in neighbors (line 16). We stop the propagation until the label value exceeds the rank of the same vertex (line 15). To look up the rank value of the vertex, we need to divide the vertex index by 2 to find out the pin index (lines 2 and 15), because we represent two transitions of the same pin as vertices with adjacent indices. Our kernel requires no other synchronization strategies because it involves no read-after-write operations. We can scan and label the constrained subgraph fairly quickly.

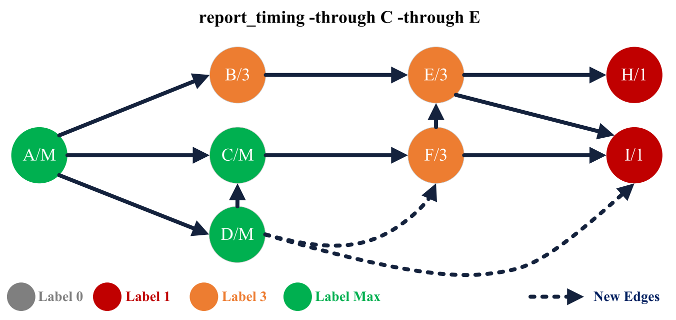


Fig. 9. Assume two additional edges exist which do not affect the ranking and scanning results. Apply the filtering rules.

C. Critical Path Filtering and Searching

In this step, we generate all critical paths that satisfy the extensive path constraints. Preprocessing steps in the previous sections help us establish the rank and label information. We reuse these information to drop the paths which fail the constraints. We enable a key filtering rule and integrate it into our path expansion iteration. Different from unconstrained path generation, we only construct suffix and prefix forests in the concerned region. Therefore, we introduce the notion of suffix subforest and prefix subforest, respectively. The fan-in edge filtering rule is defined as follows: For each fan-in edge $e_{u \to v}$

$$\text{label}(u) = \text{label}(v) \quad \text{or} \quad \text{rank}(\lfloor u/2 \rfloor) = \text{label}(v).$$

This filtering rule considers two cases. The first case checks if edge $e_{u \to v}$ connects two vertices in the same region. The second case checks if the edge is connected from the boundary region. Edge $e_{u \to v}$ is permissible under constraints if it passes either one of the tests.

To demonstrate the usage of our filtering rules, we illustrate an example in Fig. 9. We add two edges $e_{D \to I}, e_{D \to F}$ that do not affect the preprocessing results. We can first remove vertices G and J since they do not belong to the constrained subgraph. Then we apply our filtering rules to each fan-in edge. For example, fan-in edge $e_{D \to I}$ have different labels on two vertices. In other words, this fan-in edge is not connecting vertices in the same region. It also fails the second case because $\text{rank}(D) \neq \text{label}(I)$, meaning vertex D is not the boundary vertex before I. The diagram shows the only boundary vertex prior to I is E. We apply the same tests to fan-in edge $e_{D \to F}$ and know it has to be filtered as well. We ignore both edges during the path generation process.

While we enforce the filtering rules, we launch the distance relaxation kernel in Algorithm 1 to construct the suffix subforest. Distances of all endpoints are initialized with the required arrival time before the kernel launches. Whenever a fan-in edge connects to a neighbor, we apply the filtering rules to the fan-in edge. We stop the kernel launches until the distances converge.

To collect path prefix information, we iteratively construct the prefix subforest. We still enforce the filtering rules in all the GPU kernels within each iteration. Since we explore the prefix in the other direction, we define similar filtering rule for each fan-out edge $e_{u \to v}$ as well. In this way, we make sure the explored prefix does not violate any path constraint. We expand our prefix subforest in iterations until current set of explored paths has good enough accuracy or we explore all possible paths. By the end of iteration, we perform path recovery to obtain full paths considering all the given constraints.

VI. Fine-Grained Optimizations

The proposed GPU-accelerated PBA framework can significantly reduce the long runtime of the critical path generation process. However, there is still room for improving the performance of our GPU algorithms using fine-grained optimization techniques. We highlight some performance problems of our framework below:

1) *Substantial Amount of Work is Performed on the Host Side:* Host handles level compression, which is essentially an indirect sorting. Profiling shows this step becomes the execution bottleneck and takes the majority of the runtime. For example, compressing over ten million path candidates into one million paths can take above 90% of the runtime in each iteration.
2) *Per Thread Block Workload Needs to Synchronize With Each Other:* All thread blocks needs to synchronize with each other between each step. For example, the path expansion cannot start until all thread blocks complete the offset scanning step.
3) *High Communication Cost for Path Information Between Host and Device:* Path information passing between host and device has a relatively high cost compared to the kernel computation. Host needs to wait for a full set of newly expanded paths for compression, which takes over 80% of the memory communication cost. Device should wait for the compressed set of paths before expanding in the next iteration.

In this section, we propose several optimization techniques to overcome these performance limitations. These optimizations include kernel fusion with dynamic parallelism, fine-grained workload decomposition, fast merge sort strategy, and merge with dynamic parallelism. We shall demonstrate in experimental results that our optimized framework further improved the performance of our GPU-accelerated PBA framework $3-5\times$ for large circuit designs.

A. Optimization Design Overview

In this section, we describe the basic data structures and overview of our new framework. Fig. 10 shows the flow of our optimization techniques. Once a thread block is launched, all threads in each thread block will be assigned with four block-specific workload until we get blockwise-sorted critical paths. We assign the thread block to multiple parent deviation edges. Instead of simply assigning each thread to each parent deviation edge, we assign each thread to parent deviation edges that are evenly spaced in the memory. We leverage this thread coarsening strategy to reduce the overhead to spawn too many threads and maximize memory coalescing. After we obtain the compressed and sorted results from each thread block, we merge these results through a reduction tree strategy. We leverage CUDA *dynamic parallelism* to remove memory

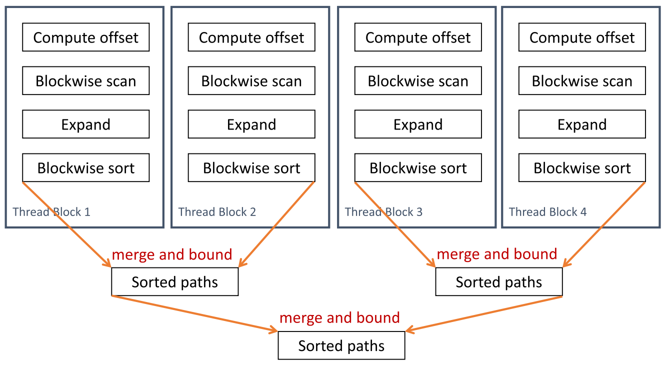


Fig. 10. Example of new GPU-accelerated PBA framework with four thread blocks.

management API and kernel calls on the host side. There are two places in our framework where we use dynamic parallelism. The first place is device memory allocation in kernel after the blockwise offset computation. The second place is reducing multiple sorted blocks into a single block. In the next section, we will describe specific optimizations that we have implemented in this optimized framework.

B. Optimization Strategies

We have implemented various optimizations for our new GPU-accelerated PBA framework. We fuse multiple steps together in a single kernel to reduce the number of kernel calls on host. We implement a fast merge sort algorithm to perform the blockwise sorting workload. In the blockwise sorting, we leverage the parallel merging inspired by Kirk and Wen-Mei [21]. We reduce blockwise-sorted results by using dynamic parallelism.

1) Kernel Fusion With Dynamic Parallelism: There are four block-specific workload. To reuse threads in each block for next step's workload, we first perform kernel fusion to steps allocate and scan, and steps expand and merge. By fusing steps allocate and scan, we compute the number of critical paths the parent path will expand, then directly save that number into shared memory, and finally perform a blockwise scan by using either Kogge and Stone [22] or Brent and Kung [23] parallel scan algorithms. By fusing steps expand and merge, we expand the child paths required by next level of path candidates, and then directly merge the expanded paths based on slack values. We effectively reduce the kernel overhead by reducing the number of necessary kernel calls. Besides, we further reduce the number of global memory accesses by converting part of the global memory access into shared memory accesses.

Based on our fusion strategy above, we have successfully reduced the number of kernel calls from four to two. We can further reduce the number of kernel calls to one by using dynamic parallelism. As shown in Listing 1, after blockwise scan, we isolate a single thread for memory allocation. We first obtain the total number of child paths in next iteration for the current block. Then we use a single thread to invoke `cudaMalloc` to allocate proper device memory of `DeviationEdge` for blockwise expansion. In this way, no host operations is involved between steps scan and expand.

```
int nextSize = shareMem[blockDim.x-1];
if(threadIdx.x == 0){
  cudaMalloc(&blockPtrs[blockIdx.x],
    nextSize*sizeof(DeviationEdge));
}
__syncthreads();
```

Listing 1. Memory allocation with dynamic parallelism.

```
merge_all_blocks<<<1, prevGridDim>>>(
    //size of each block
    deviceBlockSizes,
    //offset of each block
    deviceBlockOffsets,
    //per-block sorted slacks
    deviceSortedSlacks,
    //intrablock indices
    deviceSortedIndices,
    //interblock indices
    deviceSortedGridIndices
);
```

Listing 2. Top level kernel launch with dynamic parallelism.

Besides, threads in step expand can directly reuse the scan result in `shareMem`. We further reduce the number of global memory accesses as well. However, there are still some disadvantages when we fuse all kernels together with dynamic parallelism. The most important issue is the difficulty to debug. Because the kernel involves so much steps back to back, it is extremely hard to know if intermediate data are correctly computed. Besides, we need additional pointer arrays to save the device pointers allocated within the kernel for future memory management. Based on our experience, our recommendation is to implement kernels with separate steps first and make sure all steps can function correctly. Then an implementation with dynamic parallelism can be put into practice.

2) Block Merging With Dynamic Parallelism: Based on our previous optimization for blockwise merge sorting, we continue to merge sorted results in each block into a global set of worst critical paths. Because the length of sorted blocks may not be fully balanced, we want to concurrently launch different number of threads to merge two sorted blocks that is dynamically determined by the length of each block. CUDA dynamic parallelism can exactly satisfy this need.

At the top level, as shown in Listing 2 we launch very few parent threads, where the total thread number equals to the number of blocks in previous fused kernel. We perform indirect sorting because sorting over the entire `DeviationEdge` is costly. We use the idea of structure of arrays to keep track of path slacks, intrablock indices, and interblock indices. The reason of this is to maximize the cache utilization. We will reorder the original arrays of `DeviationEdge` later based on `deviceSortedIndices` and `deviceSortedGridIndices` to ensure minimum times of data structure movements.

At the bottom level, as shown in Listing 3, we dynamically launch children threads to merge sorted blocks in a reduction tree manner. We double `stride` for reduction access pattern and figure out the `start` and `mid` offsets for merging. We configure the grid dimension for each merge kernel based on

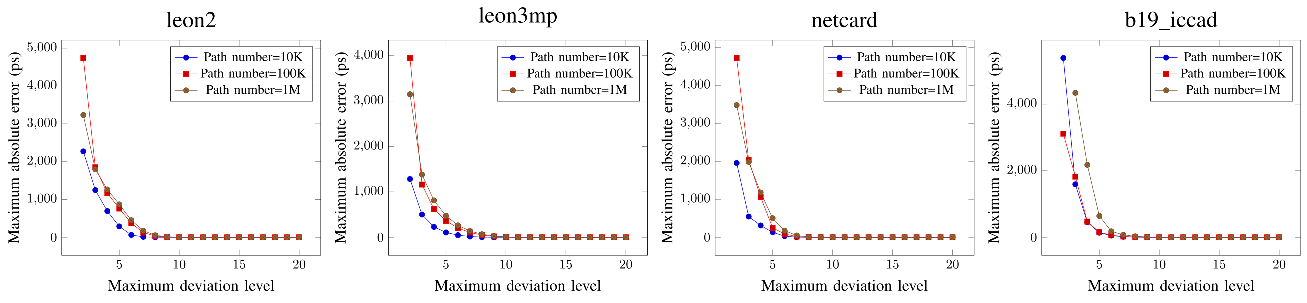


Fig. 11. Maximum absolute error between timing reports from our algorithm and OpenTimer.

```
for(int stride=2; stride<2*blockNum;
                  stride*=2){
    __syncthreads();
    int startBid = threadIdx.x * stride;
    if(startBid >= blockNum) continue;

    int start = blockOffsets[startBid];
    int midBid = startBidx + (stride/2);
    int A_len = blockSizes[startBid];
    int B_len = 0;
    int mid = start;
    if(midBid < blockNum){
        B_len = blockSizes[midBid];
        mid = blockOffsets[midBid];
    }
    int gridNum = ceil_div(A_len+B_len,
        ElePerThread*BLOCK_SIZE);
    gpu_merge_kernel
    <<<gridNum, BLOCK_SIZE>>>
    (
        &inputSlack[start], A_len, ...,
        &inputSlack[mid], B_len, ...,
        &outputSlack[start], ...
    ); //omit index arrays
    blockSizes[startBid] = A_len + B_len;

    float* temp = outputSlack;
    outputSlack = inputSlack;
    inputSlack = temp;
    ...;//index array swapping
}
```

Listing 3. Bottom level kernel launch with dynamic parallelism.

number of elements per thread and number of threads per block. After these preprocessing steps, we will dynamically launch GPU merge kernel to merge two sorted blocks. We also leverage double buffering in each iteration to reduce the memory copy costs. The final merged arrays can be generated fairly quickly with dynamic parallelism because we balance the computation resources based on the size of workload.

VII. Experimental Results

In this section, we demonstrate that our GPU-accelerated framework can efficiently generate accurate path reports. Our approach is scalable to thousands and millions of paths. We conduct our experiments on a 64-bit Ubuntu Linux machine with 1 GeForce RTX 2080 GPU and 40 2-GHz Intel Xeon Gold 6138 CPU cores. Our compiler settings are CUDA NVCC 11.0 device compiler and GNU GCC 8.3.0 host compiler, where optimization flag `-O2` and C++17 standard `-std=c++17` are enabled. In terms of kernel configuration, we use 1024 threads per block for 1-D kernel configurations and 256 $\times$ 2 threads per block for 2-D kernel configuration. We use one CPU core for all host operations. We consider the state-of-the-art path generation algorithm [4] as our baseline. To the best of our knowledge, the baseline has the best time complexity and practical efficiency. The baseline also supports critical path reporting with extensive path constraint. It has been implemented in the open-sourced STA tool, OpenTimer, as its core path generation algorithm [5], [24]. We evaluate our algorithm on real designs with a golden reference generated by an industrial standard timer [3]. To ensure fairness, we restrict our comparison to the PBA part in OpenTimer.

A. *Path Report Accuracy*

Our GPU-accelerated PBA framework can generate accurate timing report. To evaluate the accuracy, we compare separate timing reports generated from our framework and the baseline on the same number of paths. We measure the absolute slack difference for each pair of critical paths in two reports. We save the maximum absolute error and plot the error versus different number of expansion levels in Fig. 11. Our experiments scale from 10 K up to 1 M critical paths on million-gate designs. We show the summary of each circuit design in Table II. As shown in Fig. 11, our algorithm outputs almost identical critical paths as OpenTimer by expanding to ten levels on all designs. We keep generating highly accurate results even for 1 M critical paths on million-gate designs leon2 (1.6 M gates), leon3mp (1.2 M gates), and netcard (1.5 M gates). Additionally, Fig. 11 manifests tremendous accuracy improvement of our algorithm in the first few levels. For example, from level 2 to level 3, the maximum absolute error in 1 M critical paths is reduced by over 1000 ps in most million-gate designs. Our report accuracy continues to improve in the subsequent level expansion. Besides slack values, we can match the full path trace between reports. After 15 deviation levels of expansion, our framework can report 1 M critical paths with the same path trace as baseline on these benchmarks.

B. *Runtime Performance*

In this section, we demonstrate our algorithm can accelerate critical path generation process to a new milestone. We compare runtime (ms) of our algorithm (one GPU) with runtime (ms) of the PBA in OpenTimer (one CPU core) by

TABLE II
RUNTIME PERFORMANCE (MS) COMPARISON BETWEEN OPENTIMER AND OUR GPU-ACCELERATED ALGORITHM (ONE GPU)

Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	7091.79	172×	8182.84	149×
netcard	3999174	1496719	7404006	752188	2050.60	367×	2475.90	304×	2484.08	303×
vga_lcd	397809	139529	756631	53204	682.94	77.9×	683.04	77.9×	706.16	75.3×
vga_lcd_iccad	679258	259067	1243041	66582	720.40	92.4×	754.35	88.3×	766.29	86.9×
b19_iccad	782914	255278	1576198	402645	2144.67	188×	2948.94	137×	3483.05	116×
des_perf_ispd	371587	138878	697145	24120	763.79	31.6×	766.31	31.5×	780.56	30.9×
edit_dist_ispd	416609	147650	799167	614043	1818.49	338×	2475.12	248×	2900.14	212×
mgc_edit_dist	450354	161692	852615	694014	1463.61	474×	1485.65	467×	1493.90	465×
mgc_matric_mult	492568	171282	948154	214980	994.67	216×	1075.90	200×	1113.26	193×

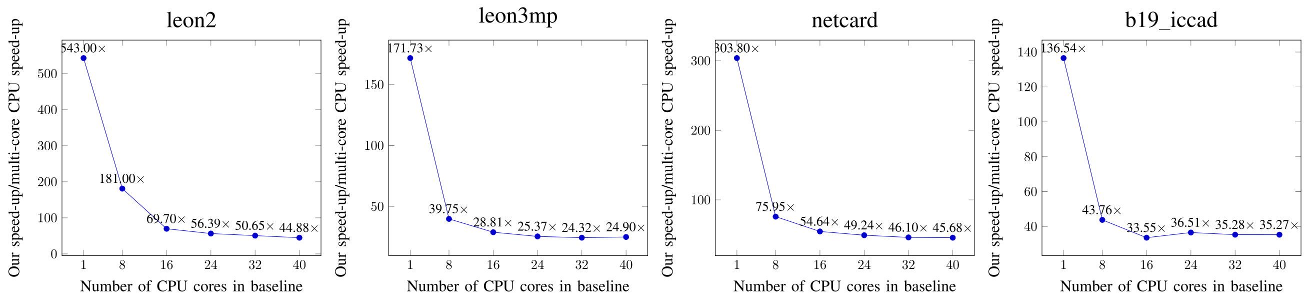


Fig. 12. Speed-up values of our algorithm over the baseline at different numbers of CPU cores.

reporting 100 K critical paths on the ten largest benchmarks. We experiment with MDL 10, 15, and 20, where we demonstrate the accuracy guarantee (see Section VII-C). Table II shows our results and the runtime comparison. We can observe that our algorithm achieves significant speed-up on million-gate designs over OpenTimer. With MDL equal to 15, we boost the baseline 543× on leon2 (1.6 M gates), 172× on leon3mp (1.2 M gates), and 304× on netcard (1.5 M gates). We also achieve over an order of magnitude speed-up on medium benchmarks, such as 77.9× on vga_lcd, 88.3× on vga_lcd_iccad, and 31.5× on des_perf_ispd.

To further demonstrate our performance advantage over the baseline, Fig. 12 plots the speed-up curve of our algorithm over the baseline across different numbers of CPU cores. We observe that the performance of baseline continues to improve as the number of cores increases but saturates at about 16 cores. We also notice there is always a significant performance margin to ours. With the baseline at the maximum CPU concurrency of 40 cores, our algorithms is still faster than the baseline by 44.88×, 24.90×, 45.68×, and 35.27× on large designs leon2, leon2mp, netcard, and b19_iccad, respectively. In fact, according to our experiments, our GPU-accelerated PBA algorithm is always faster than the baseline in all designs, regardless of the number of CPU cores the baseline uses.

C. Runtime Performance With Path Constraints

In this section, we demonstrate our GPU-accelerated PBA framework can maintain its high performance advantage when it generates critical paths under constraints. We run our framework and the baseline to report arbitrary numbers ($k = 10\,\text{K}, 5\,\text{K}, 1\,\text{K}$) of critical paths under various path constraints on the same set of designs. We verify that all generated critical paths meet the path constraints and the full path trace matches the baseline report. We summarizes the performance comparison in Table III. Our GPU-accelerated PBA framework is clearly faster than the baseline. Our framework demonstrates significant speed-up over the runtime performance on million-gate designs. For example, our framework is 53.7× and 102× faster than the baseline on designs `leon2` and `leon3mp`, respectively. Our framework exhibits promising performance advantages on medium-size designs as well. For instance, our framework achieves 12×–28× speed-ups on designs `vga_lcd`, `vga_lcd_iccad`, `des_perf_ispd`, `edit_dist_ispd`, and `mgc_edit_dist`. We can observe

TABLE III
RUNTIME PERFORMANCE (MS) CONSIDERING PATH CONSTRAINTS

Benchmark	#k	OpenTimer Runtime (ms)	CUDA PBA (1 GPU)	
			Runtime (ms)	Speed-up
leon2	10K	2.46M	45839	53.7×
leon3mp	10K	2.39M	23498	102×
netcard	5K	133K	31254	4.24×
b19 iccad	5K	180K	6993	25.8×
vga lcd	1K	40K	1997	20.0×
vga lcd iccad	1K	45K	3701	12.1×
des perf ispd	1K	53K	2073	25.6×
edit dist ispd	1K	63K	3271	19.4×
mgc edit dist	1K	104K	3766	27.5×
mgc matric mult	1K	14K	3311	4.25×

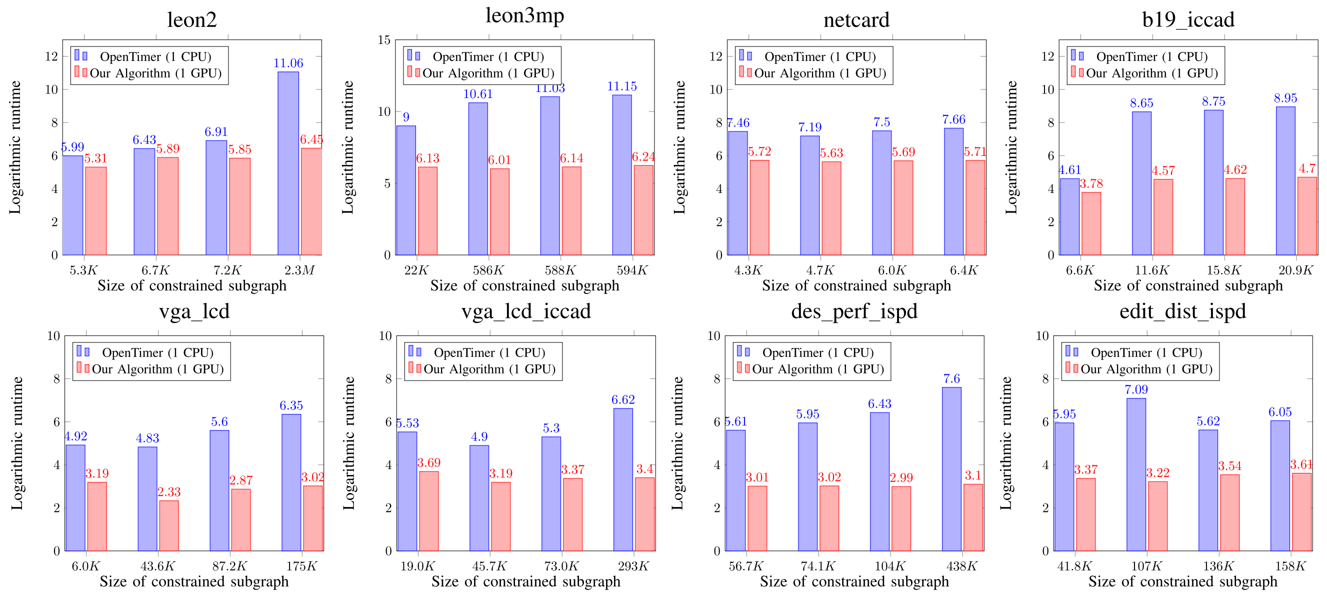


Fig. 13. Runtime comparison between OpenTimer [15] and our algorithm on different sizes of constrained subgraphs.

relatively limited speed-up in a few designs like `netcard` and `mgc_matric_mult`. This is due to highly constrained search space that limits the amount of data parallelism. We have a more in-depth discussion about the runtime performance versus size of constrained search space in the subsequent section.

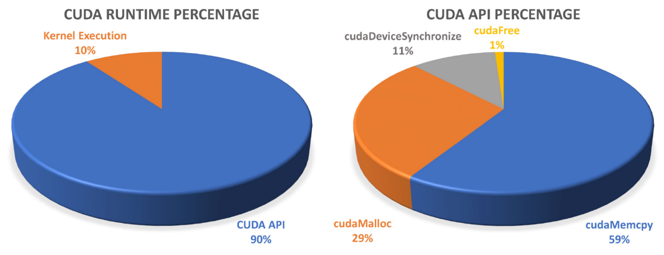
D. Runtime Versus Constrained Search Space

In this section, we discuss and analyze how the constrained subgraph size affects the runtime performance of our GPU-accelerated PBA framework when we report under path constraints. We run our framework and the baseline on designs `leon2`, `leon3mp`, `netcard`, `b19_iccad`, `vga_lcd`, `vga_lcd_iccad`, `des_perf_ispd`, and `edit_dist_ispd` under different set of path constraints. Each set of path constraints defines a different constrained subgraph. The size of the subgraph is represented by the number of pins. We plot the logarithmic runtime of our framework and the baseline versus the size of constrained subgraph in Fig. 13. In general, our framework is faster than the OpenTimer baseline at different sizes of constrained subgraph, from 4.3 K to 2.3 M. Our framework has more promising advantage on a larger constrained subgraph, because it can utilize higher data parallelism. For example, our framework has 4.61 logarithmic runtime advantage (100.5× speed-up) over the OpenTimer baseline at subgraph size of 2.3 M pins, while the advantage is much smaller for subgraphs of sizes 5.3 K–7.2 K. We can observe similar performance patterns in other designs. We revisit the `netcard` benchmark. Since all constrained subgraphs have sizes below 10 K, our framework can hardly benefit any data parallelism to achieve higher performance improvement.

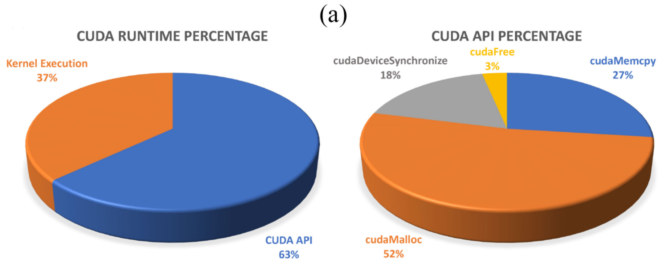
E. Performance With Fined-Grained Optimizations

We demonstrate our fine-grained optimizations can improve the performance of our GPU-accelerated PBA workload. We use the original GPU-accelerated PBA framework as baseline. As shown in Fig. 14(a), to generate 100 K critical paths in leon2, over 90% of CUDA runtime is spent on CUDA APIs, including `cudaMemcpy`, `cudaMalloc`, `cudaDeviceSynchronize`, and `cudaFree`. Within the CUDA API, 59% runtime is spent on the memory transfer or `cudaMemcpy`. More specifically, due to the high volume of path candidates before compression, memory copy from device to host takes about 82%.

With our optimizations, we successfully reduce this source of communication cost. The CUDA runtime breakdown after optimizations in Fig. 14(b) demonstrates this improvement. The GPU participates more in the computation since we move the compression step to the GPU. The communication cost due to memory transfer has dropped to 27%. We can also see the effectiveness of our optimization techniques by comparing the performance on the same set of circuit designs



(a)



(b)

Fig. 14. CUDA runtime and API breakdown comparison. (a) CUDA runtime and API breakdown of framework prior to optimizations. (b) CUDA runtime and API breakdown of framework after optimizations.

TABLE IV
RUNTIME PERFORMANCE OF PREVIOUS GPU-ACCELERATED PBA FRAMEWORK AND OPTIMIZED FRAMEWORK

Benchmark	#MDL=10			#MDL=15			#MDL=20			Peak Memory (MiB)
	Previous (ms)	Optimized (ms)	Speed-up	Previous (ms)	Optimized (ms)	Speed-up	Previous (ms)	Optimized (ms)	Speed-up	
leon2	4708.36	942.69	5.00	5295.49	1033.62	5.12	5413.84	1090.70	4.96	658
leon3mp	5520.85	944.09	5.85	7091.79	1202.89	5.90	8182.84	1395.31	5.86	590
netcard	2050.60	674.15	3.04	2475.90	731.786	3.38	2484.08	753.58	3.30	644
vga lcd	682.94	382.12	1.79	683.04	385.849	1.77	706.16	360.36	1.96	320
vga lcd iccad	720.40	382.19	1.88	754.35	385.186	1.96	766.29	384.89	1.99	348
b19 iccad	2144.67	509.28	4.21	2948.94	627.398	4.70	3483.05	740.65	4.70	360
des perf ispd	763.79	332.11	2.30	766.31	346.358	2.21	780.56	332.97	2.34	330
edit dist ispd	1818.49	473.13	3.84	2475.12	566.569	4.37	2900.14	674.51	4.30	338
mgc edit dist	1463.61	410.69	3.56	1485.65	401.359	3.70	1493.90	405.44	3.68	336
mgc matric mult	994.67	376.99	2.64	1075.90	415.331	2.59	1113.26	462.76	2.41	340

used in the previous framework. We conduct the same set of experiments on 1 GeForce RTX 2080 GPU and 2-GHz Intel Xeon Gold 6138 CPU Core. We enable dynamic parallelism in NVCC by adding the compilation flag `-rdc=true`.

A summary of experiments on the optimized framework is shown in Table IV. For each of the design circuit, we run both our framework and the baseline with expansion levels 10, 15, and 20. We first verify that our framework can correctly generate 100 K critical paths with ordered slack values. Then we document and compare the performance difference. As a comprehensive study, we also provide the peak GPU memory usage of both frameworks in the last column of Table IV. As demonstrated in Table IV, our framework can achieve 3×–5× speed-up on large designs with over 1 million gates, such as `leon2`, `leon3mp`, `netcard`, and `b19_iccad`. To gain a better understanding of the optimized framework, we also measure the runtime breakdown in the initial iterations on `leon2`, since the initial iterations have the largest numbers of deviation paths. On average, we spend 42.75% on the block-specific step, 30.82% on block merging, and the rest 26.43% on CUDA API. Besides decent speed-up on large designs, we still have relatively good improvement, 1.77×–4.70× speed-up, on the rest of medium designs. This can make a great impact to PBA acceleration, because, in `leon2` for example, we can achieve 5.00× faster on top of 611× speed-up compared to state-of-the-art CPU algorithm. We also run fine-grained optimizations considering path constraints and compare with Section VII-C. With relaxed constraints in `leon2` and `leon3mp`, we continue to obtain 3×–5× speed-up. With very strict constraints in `netcard`, the speed-up is limited due to the tiny level of deviation edges. In this set of integrated experiments, we demonstrate that we can alleviate the performance bottleneck in previous work with our new GPU-accelerated framework.

VIII. CONCLUSION

In this article, we have introduced a novel GPU-accelerated PBA algorithm to overcome the runtime bottleneck of CPU-based PBA. We decompose the critical path generation into multiple GPU-accelerated kernels and leverage the implicit path representation method to design GPU-efficient data structures. We successfully utilize the high computation throughput for large-scale path reporting on large-scale circuit designs. We also demonstrate how we can enable guided critical path reporting with extensive path constraints. We can effectively report critical paths that satisfy path constraints while maintaining high scalability. Furthermore, we propose several fine-grained optimizations on top of this GPU-accelerated PBA framework. In this new optimized framework, we overcome the performance bottleneck in host computation and communication. Experiments show that, without further optimizations, our algorithm achieves up to 543× speed-up on an 1.6 M-gate design over the state-of-the-art PBA algorithm. At the extreme, our algorithm is 25×–45× faster than the baseline of 40 cores on million-gate designs. While reporting with path constraints, we can maintain up to 100× speed-up on million-gate design. With fine-grained optimizations, we can achieve even higher performance, about 3×–5× speed-up compared to the previous framework. We believe our algorithm can promote PBA in the earlier stage of design closure flow to improve QoR and turnaround time.

REFERENCES

[1] J. Bhasker et al., *Static Timing Analysis for Nanometer Designs: A Practical Approach*. New York, NY, USA: Springer, 2009.

[2] T. Huang, M. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *Proc. ACM/IEEE DAC*, 2016, pp. 1–6.

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

[4] T.-W. Huang and M. D. F. Wong, "UI-Timer 1.0: An ultrafast path-based timing analysis algorithm for CPPR," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 11, pp. 1862–1875, Nov. 2016.

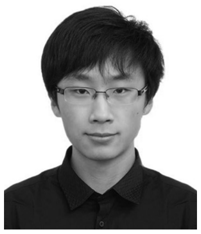
[5] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer V2: A new parallel incremental timing analysis engine," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 4, pp. 776–789, Apr. 2021.

[6] "EDA vendors should improve the runtime of PBA." 2023. [Online]. Available: https://www.electronicdesign.com/technologies/eda/article/21796368

[7] T. Huang, C. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *Proc. IEEE IPDPS*, 2019, pp. 974–983.

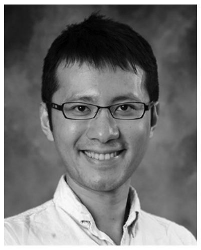
[8] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 6, pp. 1303–1320, Jun. 2022.

[9] C.-H. Chiu and T.-W. Huang, "Composing pipeline parallelism using control taskflow graph," in *Proc. ACM HPDC*, 2022, pp. 283–284.
[10] C.-H. Chiu and T.-W. Huang, "Efficient timing propagation with simultaneous structural and pipeline parallelisms: Late breaking results," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2022, pp. 1388–1389.
[11] P. Lee, I. H. Jiang, and T. Chen, "FastPass: Fast timing path search for generalized timing exception handling," in *Proc. IEEE/ACM ASPDAC*, 2018, pp. 172–177.
[12] B. Jin, G. Luo, and W. Zhang, "A fast and accurate approach for common path pessimism removal in static timing analysis," in *Proc. IEEE ISCAS*, 2016, pp. 2623–2626.
[13] F. Peng et al., "A general graph based pessimism reduction framework for design optimization of timing closure," in *Proc. ACM/IEEE DAC*, 2018, pp. 1–6.
[14] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A general cache framework for efficient generation of timing critical paths," in *Proc. IEEE/ACM DAC*, 2019, pp. 1–6.
[15] G. Guo, T. W. Huang, C. X. Lin, and M. Wong, "An efficient critical path generation algorithm considering extensive path constraints," in *Proc. IEEE/ACM DAC*, 2020, pp. 1–6.
[16] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated path-based timing analysis," in *Proc. IEEE/ACM DAC*, 2021, pp. 721–726.
[17] G. Guo, T.-W. Huang, Y. Lin, and W. Martin, "GPU-accelerated critical path generation with path constraints," in *Proc. IEEE/ACM ICCAD*, 2021, pp. 1–9.
[18] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. IEEE/ACM SC*, 2009, pp. 1–11.
[19] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. HiPC*, 2007, pp. 197–208.
[20] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA solutions for the SSSP problem," in *Proc. ICCS*, 2009, pp. 904–913.
[21] D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-on Approach*. Boston, MA, USA: Morgan Kaufmann, 2016.
[22] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.
[23] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.
[24] T. Huang and M. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM ICCAD*, 2015, pp. 895–902.



Guannan Guo received the B.S. degree from the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2017, where he is currently pursuing the Ph.D. degree with research focus on circuit timing analysis, heterogeneous and parallel computing, distributed systems, and scheduling algorithms.

Mr. Guo is a recipient of the Best Paper Awards at TAU 2021 Timing Workshop.



Tsung-Wei Huang (Member, IEEE) received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University, Tainan, Taiwan, in 2010 and 2011, respectively, and the Ph.D. degree from the Electrical and Computer Engineering (ECE) Department, University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2017.

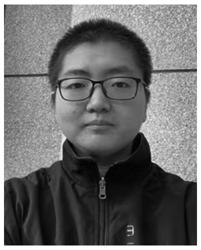
He is currently an Assistant Professor with the ECE Department, University of Utah, Salt Lake City, UT, USA. He has been researching novel software systems to streamline the building of high-performance computing applications.

Dr. Huang is the recipient of the 2019 ACM/SIGDA Outstand Ph.D. Dissertation Award and the NSF CAREER Award.

Yibo Lin (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiaotong University, Shanghai, China, in 2013, and the Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, TX, USA, in 2018, advised by Prof. D. Z. Pan.

He worked as a Postdoctoral Researcher with the University of Texas at Austin from 2018 to 2019. He is currently an Assistant Professor with the Department of Computer Science, Peking University, Beijing, China, associated with the Center for Energy-Efficient Computing and Applications. His research interests include physical design, machine learning applications, emerging technology in VLSI CAD, and hardware security.

Dr. Lin is a recipient of the Best Paper Awards at DAC 2019, *Integration, the VLSI Journal* 2018, and SPIE Advanced Lithography Conference 2016.



Zizheng Guo received the B.S. degree in computer science from Peking University, Beijing, China, in 2022, where he is currently pursuing the Ph.D. degree with the School of Integrated Circuit.

He is currently working on static timing analysis and power analysis problems in VLSI CAD. His research interests include data structures, algorithm design, and GPU acceleration for combinatorial optimization problems.

Mr. Guo is the First Place Winner of the 2022 ACM Student Research Competition Grand Finals.

Sushma Yellapragada is currently pursuing the graduate degree with the Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL, USA, where she primarily studies high-performance computing, parallel computing, and computer architecture.

She has prior experience contributing to open-source HPC technologies.

Martin D. F. Wong (Fellow, IEEE) received the B.S. degree in mathematics from the University of Toronto, Toronto, ON, Canada, in 1979, and the M.S. degree in mathematics and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign (UIUC), Champaign, IL, USA, in 1981 and 1987, respectively.

From 1987 to 2002, he was a Faculty Member of Computer Science with the University of Texas at Austin, Austin, TX, USA. He returned to UIUC in 2002, where he was the Executive Associate Dean from 2012 to 2018 for the College of Engineering, and the Edward C. Jordan Professor of Electrical and Computer Engineering. He is currently the Dean of the Faculty of Engineering, Chinese University of Hong Kong, Hong Kong.

Dr. Wong is a Fellow of ACM.