GPU-accelerated Critical Path Generation with Path Constraints

Guannan Guo*, Tsung-Wei Huang†, Yibo Lin‡, and Martin Wong*§
*Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA
†Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT, USA
‡Department of Computer Science, Peking University, Beijing, China
§Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

***Abstract*—Path-based Analysis (PBA) is a pivotal step in Static Timing Analysis (STA) for reducing slack pessimism and improving quality of results. Optimization flows often invoke PBA repeatedly with different critical path constraints to verify correct timing behavior under certain logic cone. However, PBA is extremely time consuming and state-of-the-art PBA algorithms are hardly scaled beyond a few CPU threads under constrained search space. In order to achieve new performance milestone, in this work, we propose a new GPU-accelerated PBA algorithm which can handle extensive path constraints and quickly report arbitrary number of critical paths in constrained search space. Experimental results show that our algorithm can generated identical path report and achieve up to 102× speed up on a million-gate design compared to the state-of-the-art algorithm.**

I. Introduction

Path-based Analysis (PBA) is an important step in Static Timing Analysis (STA) for reducing slack pessimism and improving quality of results [1]. However, PBA is also well-known for its extremely high runtime complexity. PBA can be 10-1000× slower than Graph-based Analysis (GBA) [2], [3]. As a key routine in PBA, critical path generation takes a significant amount of time and it is very challenging to parallelize. Existing works closely rely on CPU parallelism to accelerate the process of critical path generation [4], [5], [6], [7], [8], [9]. Nevertheless, CPU-based parallel approaches do not scale beyond a few CPU cores. For example, speed-up of state-of-the-art PBA algorithm [4], [6] stagnates at 16 CPU cores.

This problem becomes more challenging when path constraints are considered during the generation process. Optimization flows often invoke PBA repeatedly with different path constraints to verify correct timing behavior under certain logic cone. For instance, Figure 1 illustrates an example where the timer needs to report critical paths passing through `Inst1/A1` and then `Inst4/Zn`. Two paths satisfying this constraint are marked as green and red. Path constraints limit the search space and make the state-of-the-art path generation algorithms almost sequential. There are a few existing works that focused on critical path generation under constraints [10], [11], but both of them are mainly sequential algorithms and target at CPU architectures. Lai [10] proposes a cache framework which intends to save runtime from repeated constraints in multiple requests, but it has little improvement to each single request. Guo [11] introduces a faster sequential algorithm for a single timing request, but it has limited parallelization benefit. Both of them try to improve runtime performance on top of their frameworks by CPU multi-thread parallelization. However, they can only achieve 3× speed-up at maximum. In most of the time, constrained subgraphs can still contain hundreds of thousands of pins and the number of requested paths can be thousands as well. Existing speed-up by CPU parallelism is far from desirable.

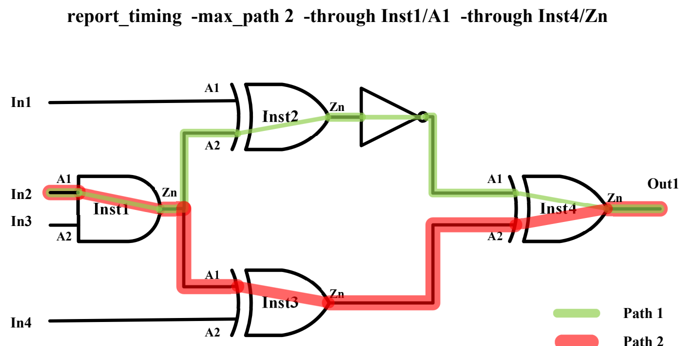


Fig. 1: An example of report timing request with path constraints.

Therefore, we seek new solutions to accelerate the generation of critical paths with path constraints by leveraging the power of Graphic Processing Unit (GPU) parallelism. However, it is a very challenging task to generate critical paths with constraints using GPU, mainly due to the following three reasons. Firstly, we need to construct GPU-efficient data structures to identify constrained subgraphs from an STA graph and filter out unwanted paths. Secondly, since critical path generation is a highly dynamic process, we need a specialized path search method that utilizes the power of data parallelism on GPU. Lastly, we must combine the generation of critical paths with path filtering to maximize the performance of GPU kernels. To overcome these challenges, in this work, we propose a GPU-accelerated critical path generation algorithm considering path constraints. We highlight three key contributions of our work as follows:

- **Efficient Filters for Constraint Satisfaction**. Our critical path filters are array-based and easy to manage on GPU architectures. We construct a special ranking array to replace topological orders and a scanning array to label regions in a constrained subgraph. By comparing values within these

two arrays, we can quickly identify if critical path candidates satisfy the constraints.

- **Data-oriented Path Verification and Generation**. We design our critical path searching and filtering for high data throughput. Multiple GPU threads can update different memory locations simultaneously in our filtering arrays construction. During the path search phase, we dispatch GPU threads to explore new path candidates and verify if each new path candidate passes values comparison in our filter at the same time.
- **High Scalability on Large Constrained Subgraphs**. Our path search strategy enables thousands of GPU threads to search for candidate critical paths simultaneously. Since path constraints are translated into array-based filters, all active GPU threads will participate in searching and filtering by referring to different memory locations of the arrays. It has high parallelization benefit especially for large constrained subgraphs.

We evaluate our algorithms on real designs generated by an industrial standard timer [12]. We randomly select common pin sequences in full timing report as path constraints so that we can test on various constrained subgraphs. Our algorithm can generate critical paths that fully satisfy path constraints. Furthermore, path traces in our timing report exactly match the report generated by the state-of-the-art algorithm [4], [11]. Compared with the baseline, we achieve up to 102× speed-up on a large design of 1.6 million gates. We also provide an SYCL implementation of our algorithm to study the advantage of single-source heterogeneous parallelism and performance portability. By running our algorithm on CPU with the same source code in SYCL, we can still achieve $2.59\times$ overall speed-up over the baseline on benchmarks with large constrained subgraphs.

II. Path-based Timing Analysis

PBA is a pivotal step in STA [1]. In STA, we abstract the design circuit as a *Directed Acyclic Graph* (DAG) $G = \{V, E\}$. Each pin in the circuit is modeled as a vertex in the STA graph. A pin-to-pin connection is modeled as an edge. Edge directions are oriented so that the STA graph flows from inputs to outputs. We use an ordered sequence of vertices or edges to represent a path. In STA, we measure the amount of timing violation by taking the difference between signal arrival time and signal required arrival time. This amount of timing violation is often denoted as *slack*. If slack is negative, STA has to report the path causing the violation as a critical path. GBA often happens before PBA to propagate timing information across the STA graph, including slew, delay, arrival time, and required arrival time. Critical paths can be identified in the phase of GBA. However, since GBA assumes worst-cast scenarios during timing updates, PBA is needed to reduce pessimism with path-specific updates such as common path pessimism removal (CPPR) and advanced on-chip variation (AOCV). Therefore, the generation of critical paths is a very important routine in PBA. Since the number of paths can be exponential to the graph size, the critical path generation process can take an extremely long time. Additionally, path constraints also complicate the generation process.

III. Critical Path Constraints

Critical path constraints define the rules that each path has to follow. These rules can be categorized into the following types:

- **-max_path**: The timer reports an arbitrary number of critical paths. The number is specified by this parameter. We often denote this number as k as well. Different from previous work that only tests on $k < 32$ [10], [11], we aim at better performance for a larger number of critical paths.
- **-from**: This parameter specifies the path starting point. It can be combined with an optional transition (rise or fall) , if a transition starting from this pin is required.
- **-through**: Each occurrence of this parameter specifies a pin that the critical path must pass through. A sequence of these parameters means the critical path must pass through these pins in sequence. Figure 1 gives an example usage.
- **-to**: This parameter specifies critical path endpoint. A optional transition can also be specified.

These are the most important types of constraints that define the constrained subgraph. There are other constraints like `-split`, which requires an analysis mode (min or max).

IV. Proposed Algorithm

Figure 2 shows an overview of our critical path generation algorithm considering path constraints. The majority of the steps are executed on GPU. We use CPU for algorithm flow control and result collection. Our algorithm begins with global ranking, where each vertex is assigned with a rank value based on the vertex's connectivity. Then we use the rank values and sequence in path constraints to scan the STA graph. The constrained subgraph is labeled and used for later steps. We leverage the implicit critical path representation [4] in our path search strategy. We construct a suffix sub-forest that contains all path suffix information in the constrained subgraph. Then we explore critical path candidates by alternating path prefix and save the prefix information in a prefix sub-forest. Details and explanations of each step are given in the subsequent sections.

A. STA Graph Structure on GPU

As the first step to generate critical paths on GPU, we need efficient data structures to represent the STA graph. We take the collection of all fan-in or incoming edges of each vertex and denote it as graph $G^- = \{V, E^-\}$. Similarly, we take the collection of all fan-out or outgoing edges of each vertex and denote it as graph $G^+ = \{V, E^+\}$. The graph representations G^- and G^+ are saved on GPU in the Compressed Sparse Row (CSR) format. CSR is one of the most common graph formats used in GPU applications [13]. CSR requires three 1D arrays to represent a weighted directed graph. The format includes a vertex array for row offsets, an edge array for column values,

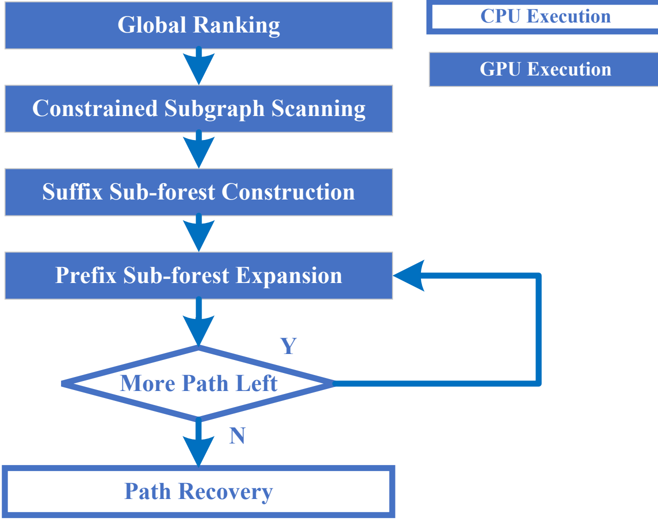
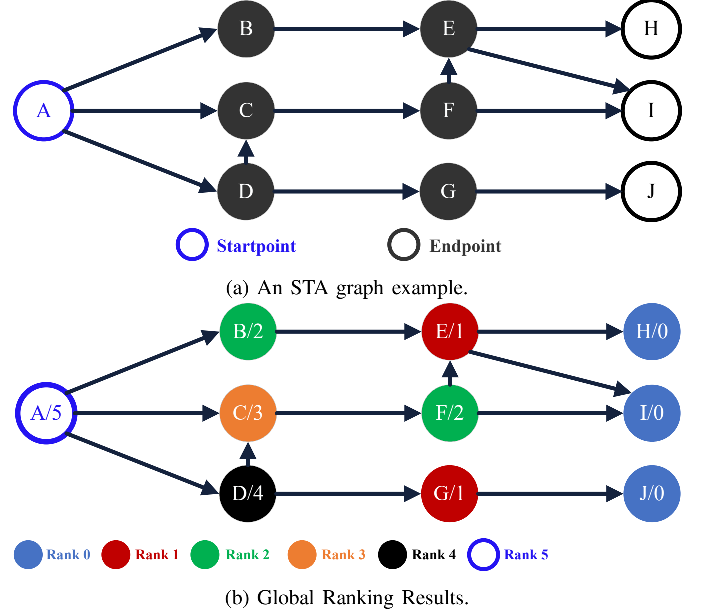


Fig. 2: Overview of our GPU-accelerated critical path generation algorithm with path constraints.



(a) An STA graph example.

(b) Global Ranking Results.

Fig. 3: Global ranking on GPU.

and a weight array for weights of all edges. This array-based designs make CSR very memory-efficient. An STA graph with vertex number N and edge number M can be represented in CSR with memory complexity $N + 2M$.

B. GPU-Accelerated Ranking

To quickly verify connectivity between two vertices, we propose a global ranking strategy to replace topological sort. The rank value represents the maximum number of edges to the graph endpoints. Therefore, multiple vertices can share the same rank values. This ranking strategy preserves the property of topological orders. For any edge $e_{u \to v} \in E$, $rank(u) > rank(v)$. Figure 3 shows an example of how we rank vertices in Figure 3a and obtain rank values in Figure 3b.

All endpoints, H, I, and J, in the STA graph, are initialized with rank value 0. Then rank values are propagated from endpoints to startpoints. We notice that vertex F is adjacent to the endpoint I, but $rank(F)$ is 2 because it has a longer path $F \to E \to H$. Compared to topological orders, the advantage of our method is less synchronization. We do not need to assign unique order value to each vertex. All GPU threads can freely propagate their rank values to corresponding neighbor pins.

Our rank propagation kernel is shown in Algorithm 1. We launch a 2D kernel where each thread in the x axis corresponds to a pin (line 1) and each thread in the y axis corresponds to a rise or fall transition. We choose this kernel configuration because each pin is mapped to two vertices in an STA graph, representing the same pin under two transitions. In our memory layout, these two vertices are saved in adjacent memory locations (line 3). For all the GPU threads that have received a rank update (line 7), we propagate their rank values plus one to their neighbors (line 15 and 16). All the neighbor received new rank values validate the update by comparing with their old ranks. We see that all threads are writing into multiple memory locations and the only synchronization

Algorithm 1: Propagate Rank Kernel

Input : G^{-}in CSR format, N as #vertices, M as #edges, vertices$[N]$, edges$[M]$, weights$[M]$
Input : Previous rank values, ranks$[N/2]$
Input : Rank cache, rankCache$[N/2]$
Input : Array indicating vertices with updated ranks, rankUpdated$[N/2]$
Result: Rank values array, ranks$[N/2]$

1 pinId ← **blockIdx**.x ∗ **blockDim**.x + **threadIdx**.x;
2 riseFall ← **threadIdx**.y;
3 tid ← 2*pinId + riseFall;
4 **if** *tid* $\geq N$ **then**
5 | **return**;
6 **end**
7 **if** *rankUpdated[pinId]* **is false then**
8 | **return**;
9 **end**
10 rankUpdated[pinId] ← **false**;
11 edgeStart ← vertices[tid];
12 edgeEnd ← (tid == N-1) ? M : vertices[tid+1];
13 **for** *eid ← edgeStart to edgeEnd* **do**
14 | neighborPin ← edges[eid]/2 ;
15 | newRank ← ranks[pinId] + 1;
16 | **atomicMax** (&rankCache[neighborPin], newRank);
17 **end**
18 **return**;

happens in the atomic operation (line 16). Atomic operations are distributed over different memory locations so we can expect less thread contention compared to topological sort. Since multi-threading topological sort requires sychronization over a single counter, it introduces a higher overhead.

C. Constrained Subgraph Scanning

At this step, we use GPU kernels to scan and label the subgraph defined by path constraints. Moreover, based on connectivity to the sequence of pins in path constraints,

different labels are assigned to vertices. All the labels are saved in a 1D array `logicCone[N]` for memory efficiency. Before we launch our kernel, we perform the following initialization. For each pair of consecutive pins u, v appear in the path constraints, the label of vertex associated with latter pin v is the rank value of the previous pin u. For instance, if `-through u -fall_through v` appear in the constraints, where u, v already denote their pin indices, we have `logicCone[2*v+1] = rank[u]`. For the first pin in the constraints, we simply assign `INT_MAX` as label.

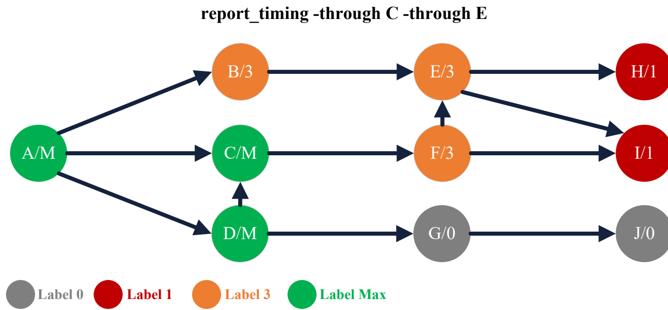


Fig. 4: Subgraph scanning results.

Figure 4 shows the subgraph scanning results given the STA graph in Figure 3a and path constraints `-through C -through E`. To simply this example, we do not consider transitions. A pin is equivalent to a vertex in the graph. All labels are initialized with zeros. Vertices from endpoints up to the last `through` pin E (not including) get labels equal to the rank of last `through` pin. Therefore, H and J get label values 1. Vertices from E up to C (not including) get labels as $rank(C) = 3$. Vertices C and upwards get labels `INT_MAX`. Vertices G and J are not connected to the last `through` pin, so their labels remain 0.

We scan the STA graph with the subgraph scanning kernel shown in Algorithm 2. The idea of this subgraph scanning kernel is to propagate labels until the label value exceeds the rank value of the neighbor pin. We launch this kernel with 1D configuration. Each thread is assigned with a vertex in the STA graph (line 1). For the vertex that has recently received a label update (line 5), propagate the same label value to its neighbors (16). The propagation stops when the label exceeds the rank value of the neighbor pin (line 15). Since there are no read-after-write operations, no atomic operations are required. The subgraph can be quickly scanned and labeled. All the non-zero entries define the size of our constrained subgraph.

D. Critical Path Filtering and Searching

The goal of this step is to identify all critical paths satisfying the constraints and order them based on their criticality. After the preprocessing steps in the previous sections, we can quickly filter paths that fail to satisfy the path constraints. We only have to integrate our filters with the path generation process. Our critical path generation relies on two complementary data structures, suffix forest, and prefix forest. Since both of the forests are constructed within the constrained subgraph, we denote them as suffix sub-forest and prefix sub-forest. As shown in Figure 2, the suffix sub-forest is fully built for once. Then we iteratively grow prefix sub-forest until we reach the maximum number of paths required or no more paths can be explored. In the following sections, we first explain a key definition in our path generation algorithm. Then we show how each sub-forest is constructed in details.

Algorithm 2: Scan Subgraph Kernel

```
Input  : G⁻ in CSR format, N as #vertices, M as #edges,
         vertices[N], edges[M], weights[M]
Input  : Rank values, ranks[N/2]
Input  : Previous label values, logicCone[N]
Input  : Label cache, logicConeCache[N]
Input  : Array indicating vertices with updated labels,
         labelUpdated[N]
Result: Label array, logicCone[N]
1  tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
2  if tid ≥ N then
3  |   return;
4  end
5  if labelUpdated[tid] is false then
6  |   return;
7  end
8  prevRank ← logicCone[tid];
9  labelUpdated[pinId] ← false;
10 edgeStart ← vertices[tid];
11 edgeEnd ← (tid == N-1) ? M : vertices[tid+1];
12 for eid ← edgeStart to edgeEnd do
13 |   neighbor ← edges[eid] ;
14 |   neighborPin ← neighbor/2 ;
15 |   if ranks[neighborPin] < prevRank then
16 |   |   logicConeCache[neighbor] ← prevRank;
17 |   end
18 end
19 return;
```

1) Implicit Path Representation: We leverage the idea of implicit path representation [4] for efficient memory usage on GPU. Each path is separated into two complementary parts, a prefix, and a suffix. For a single endpoint, all suffix information are saved in a suffix tree. The suffix tree is simply the shortest path tree. The prefix tree, on the other hand, contains all edges not in the suffix tree. These edges are also defined as *deviation edges*. In the prefix tree, each tree node is a deviation edge and it implicitly represents a path. For instance, Figure 5 shows how path $< e_3, e_8, e_{11}, e_{14} >$ can be implicitly represented by the tree node e_{11} in the prefix tree. Since e_{11} has no parents, it means e_{11} is the only deviation edge in the path. By complementing with edges from the suffix tree (e_3, e_8, and e_{11}), we can obtain the full path trace. Readers can refer to [4] for more details on suffix-prefix representation.

2) Suffix Sub-forest Construction: The state-of-the-art algorithm [4] constructs individual pair of suffix trees and prefix trees with respect to each endpoint. However, the constrained subgraph often contains multiple endpoints. Such individual tree construction wastes both runtime and memory because it is common for these endpoints to share logic cones. Therefore, we introduce the notion of suffix sub-forest by unifying all

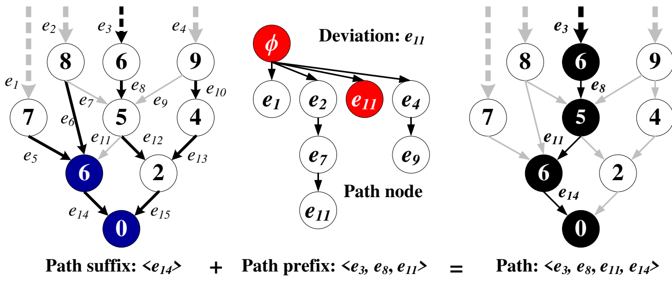


Fig. 5: Example of implicit path representation [4].

suffix tree at different endpoints. To make sure all path suffix follow the constraints as well, we use the following filtering rules when we traverse each fan-in edge. For each fan-in edge $e_{u \to v}$,

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

The first assertion checks if the neighbor vertex u share the same label as v. This assertion verifies if edge $e_{u \to v}$ stays in the same region. The second assertion verifies if the neighbor vertex u is the boundary to the previous region. Passing either one of the assertions suggests edge $e_{u \to v}$ is permissible under constraints. Besides, a non-zero label proves that the vertex is in the constrained subgraph.

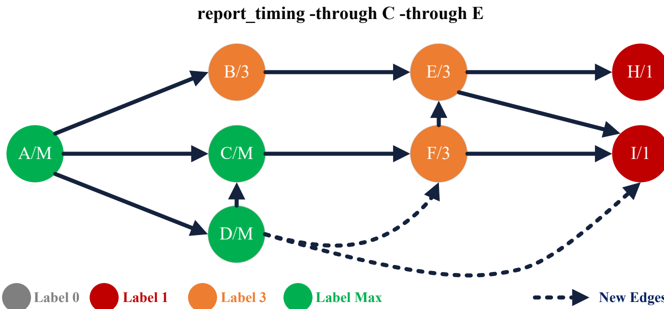


Fig. 6: Assume two additional edges exist which do not affect the ranking and scanning results. Apply the filtering rules.

Given the same STA graph and path constraints from Figure 4, we assume there are two additional edges $e_{D \to I}, e_{D \to F}$ which do not affect ranking and scanning results. We use Figure 6 to demonstrate our filtering rules. First vertices G and J can be removed because they are labeled as 0. Then for each of the additional edges, we apply the filtering rules. For fan-in edge $e_{D \to I}$, $label(D) \neq label(I)$ means they are not in the same region. We check the other rule: $rank(D) \neq label(I)$ suggests that D is not the boundary vertex before I. By looking at the diagram, we know that the only boundary vertex before I is E. Therefore, edge $e_{D \to I}$ has to be filtered. Apply same rules again to $e_{D \to F}$, we have $label(D) \neq label(F)$ and $rank(D) \neq label(F)$, so $e_{D \to F}$ has to be filtered as well. Both edges do not belong to the constrained subgraph because they fail the filtering rules.

Since the filtering rules have been established, we enforce these rules in our distance relaxation kernel. The main body of the kernel is very similar to the rank propagation kernel showed in Algorithm 1 except that we are propagating minimum distance instead of maximum ranks. Before the distance relaxation kernel launches, we initialize distances of all endpoints with the required arrival time. Inside the kernel, all vertices propagate their latest distances plus edge weights to neighbors. Whenever a new neighbor is explored, we verify the neighbor with the filtering rules. Since multiple threads may read and write to the same memory location, we use atomic operations to synchronize. The kernel terminates until no more distance relaxation is possible. The fan-in edge that contributes to the latest relaxation is saved as an edge in the suffix sub-forest. Since we allow multiple threads to perform updates concurrently, the suffix sub-forest can be constructed efficiently.

TABLE I: Data field of *deviation edge*

Field	Definition
`level`	deviation level number
`from`	deviation starting point
`to`	deviation ending point
`parent`	parent index from previous level
`childOffset`	row offset of children in next level
`slack`	critical path slack value

3) Prefix Sub-forest Expansion: After suffix sub-forest is constructed, we explore new critical paths by expanding prefix sub-forest iteratively. In the implicit path representation, we use *deviation level* (or just level) to denote the number of deviation edges in the full path trace. In our algorithm, we expand all critical paths in the same level at each iteration. These critical paths in the same level are maintained in an 1D array. Table I shows an overview of data fields for each deviation edge entry in the array. To set up the connection between adjacent levels, we use `parent` to trace back to parent entry in the previous level. Field `childOffset` act as a pointer to child paths in the next level.

In each iteration, we facilitate the deviation level expansion by multiple GPU kernels. In all GPU kernels, the filtering rules are still enforced. We use identical filtering rules for each fan-out edge $e_{u \to v}$.

The first GPU kernel we launch is the offset calculation kernel. The goal of this kernel is to set up the `childOffset` field. Each thread is assigned to a deviation edge in the current level. By traversing along the path represented by this deviation edge, the thread can calculate the number of child paths by accumulating a deviation counter at each vertex. A prefix-sum kernel is immediately followed so that the count of child paths is turned into the correct offset. After the prefix-sum is complete, `childOffset` of the last entry is the size of the next deviation level. We use this size to allocate space for the next deviation level.

When the allocation completes, we launch our second kernel which fills data into data fields of the next deviation level. Details of this kernel are shown in Algorithm 3. Similar to the first kernel, we dispatch each thread a deviation edge at

```
Algorithm 3: Expand Prefix Sub-Forest Kernel
Input  : G+ in CSR format, N as #vertices, M as #edges,
         vertices[N], edges[M], weights[M]
Input  : Rank array, rank[N/2]
Input  : Label array, logicCone[N]
Input  : Suffix sub-forest, forest[N] as edge array,
         distances[N] as distance array
Input  : currLevel as current deviation level
Input  : levelSize as the number of entries in current
         deviation level
Result: Explore critical path candidates in next level
1  tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
2  if tid ≥ levelSize then
3  |  return;
4  end
5  offset ← (tid == 0) ? 0 : currLevel[tid-1].childOffset;
6  level ← currLevel[tid].level;
7  slack ← currLevel[tid].slack;
8  v ← currLevel[tid].to;
9  while v is not endpoint do
10     pinId ← v/2 ;
11     prevRank ← logicCone[v];
12     edgeStart ← vertices[v];
13     edgeEnd ← (v == N-1) ? M : vertices[v+1];
14     for eid ← edgeStart to edgeEnd do
15         neighbor ← edges[eid];
16         neighborPin ← neighbor/2;
17         if logicCone[neighbor] > 0 and
            (logicCone[neighbor] == prevRank or
            ranks[pinId] == logicCone[neighbor]) then
18             weight ← weights[eid];
19             if eid is deviation edge then
20                 /* Fill out child path data */
21                 newPath ← nextLevel[offset];
22                 newPath.level ← level+1;
23                 newPath.from ← v;
24                 newPath.to ← neighbor;
25                 newPath.parent ← tid;
26                 newPath.childOffset ← 0;
27                 newPath.slack ← slack +
                     distances[neighbor] + weight -
                     distances[v];
28                 offset ← offset + 1;
29             end
30         end
31     end
32     /* Traverse along the suffix sub-forest
           */
33     v = forest[v];
34 end
35 return;
```

the current level (line 1). The thread continues to search for deviations starting from the place left off by the parent (line 8). For each vertex along the current path (line 9 and 33), we iterate through all the neighbors (line 14). For neighbors not in the constrained subgraph or associated fan-out edge failing filtering rules (line 17), we ignore these neighbors. For neighbors not belonging to the suffix sub-forest (line 19) and passed the filtering rule, we fill in the child deviation information (line 21-28). This expansion kernel can operate in high throughput, because each thread writes into disjoint memory locations and workload is balanced between threads.

With the help of filtering rules, all generated critical paths satisfy the path constraints. However, we may expand more paths than necessary and we need to order these paths based on slack. We choose to perform an indirect sort and compress the newly expanded level.

We continue this iteration of expansion until the desired number of critical paths are collected or no more paths are available. We merge the critical path candidates from all deviation levels and perform path recovery to obtain the final report. An example of path recovery is shown in Figure 6. For each critical path represented as a deviation edge, we first use `parent` to collect all deviation edges along the path. Then we combine the deviations edges with complementary edges in the suffix sub-forest to obtain the full path trace.

V. Experimental Results

In this section, we demonstrate that our GPU-accelerated critical path generation algorithm can report critical paths with constraints at a promising speed. We conduct our experiments on a 64-bit Ubuntu Linux machine with one GeForce RTX 2080 GPU and one 2GHz Intel Xeon Gold 6138 CPU core. For CUDA compilation, we use CUDA NVCC 11.0 device compiler and GNU GCC 8.3.0 host compiler. In addition to CUDA, we implement our algorithm using *SYCL* with Intel oneAPI DPC++ compiler (clang++ v13) [14] to study the advantage of *single-source* parallelism. SYCL has emerged as an important tool to program heterogeneous parallelism using completely standard C++ [15]. Unlike CUDA, SYCL allows programmers to write single-source C++ that offloads kernels to any SYCL or OpenCL devices, such as CPUs, GPUs, and FPGAs. This design enables a unified programming environment for EDA workloads to incorporate new heterogeneous parallelism without wrangling with different codebases. For both implementations, we enable optimization flag `-O2` and C++17 standard `-std=c++17`. We use 512×1 threads per block for 1D kernel configuration and 256×2 threads per block for 2D kernel configuration.

A. Baseline

We choose the state-of-the-art critical path generation algorithm with path constraints [4], [11] as our baseline. To the best knowledge of the authors, the baseline has the best time complexity and practical efficiency; also, it has been implemented in the open-sourced STA tool, OpenTimer, as its core path generation algorithm [16], [17], [18]. We evaluate our algorithm on real designs generated by an industrial standard timer [12]. We collect common pin patterns appearing in a full timing report and randomly select common patterns as path constraints. As shown in [11], using one CPU core is faster enough in most cases due to constrained search space, and increasing the number of CPU cores does not provide much performance benefit. We compare our GPU algorithm with the baseline on one CPU core. We report the entire runtime instead of specific steps because majority of the time is spent on suffix and prefix forests construction.

TABLE II: Runtime performance (ms) comparison between OpenTimer and our critical path generation algorithm

Benchmark	#Pins	#Gates	#Nets	#Arcs	#k	OpenTimer Runtime	CUDA PBA (1 GPU)		SYCL PBA (1 GPU)	
							Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	1616984	7984262	10K	2460260ms	45839ms	53.7	44111ms	55.8
leon3mp	3376821	1247725	1247979	6277562	10K	2391060ms	23498ms	102	23220ms	103
netcard	3999714	1496719	1498555	7404006	5K	132654ms	31254ms	4.24	27830ms	4.77
b19_iccad	782914	255278	255300	1576198	5K	180220ms	6993ms	25.8	6538ms	27.6
vga_lcd	397809	139529	139635	756631	1K	40019ms	1997ms	20.0	2150ms	18.6
vga_lcd_iccad	679258	259067	259152	1243041	1K	44792ms	3701ms	12.1	3732ms	12.0
des_perf_ispd	371587	138878	139112	697145	1K	53110ms	2073ms	25.6	2381ms	22.3
edit_dist_ispd	416609	147650	150212	799167	1K	63448ms	3271ms	19.4	3114ms	20.4
mgc_edit_dist	450354	161692	164254	852615	1K	103600ms	3766ms	27.5	3706ms	27.9
mgc_matric_mult	492568	171282	174484	948154	1K	14071ms	3311ms	4.25	3247ms	4.33

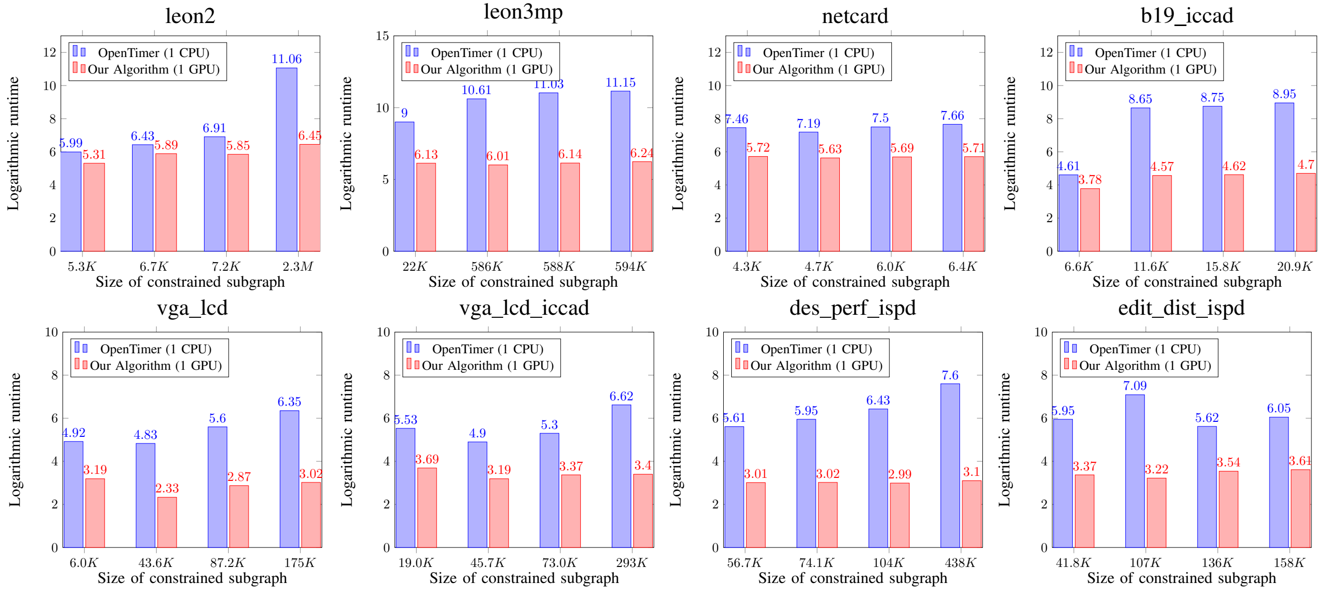


Fig. 7: Runtime comparison between OpenTimer [11] and our algorithm on different sizes of constrained subgraphs

B. Runtime Performance

Our algorithm can quickly generate critical paths that satisfy path constraints. Table II shows an overview of our experiments. We use OpenTimer and our algorithm to generate arbitrary numbers ($k = 10K, 5K, 1K$) of critical paths on different designs. We first verify that full path traces in both timing reports are identical. Given identical reports, we compare runtime performance of OpenTimer with both CUDA and SYCL versions of our algorithm on one GPU. We see that CUDA and SYCL implementations have similar performance. Both of them have a clear runtime advantage over the baseline. We can observe a significant speed-up on million-gate designs. For instance, we achieve 53.7× speed-up on `leon2` of 1.6M gates and 102× speed-up on `leon3mp` of 1.2M gates. The performance benefit of our algorithm for medium-size designs is also pronounced. We obtain 12–28× speed-ups on designs `vga_lcd`, `vga_lcd_iccad`, `des_perf_ispd`, `edit_dist_ispd`, and `mgc_edit_dist`. For some designs, the performance benefit of our algorithm is limited to the search space that constrains data parallelism. For instance, we achieve 4.24× speed-up on `netcard` and 4.25× speed-up on `mgc_matric_mult` over the baseline, which are less than an order of magnitude. In these designs, critical paths are generated within a very constrained subgraph with size less than $10K$. A more detailed analysis is given in the next section. As for memory usage, our algorithm overcomes the GPU capacity challenge by using highly compact data structures such as CSR graph format and implicit path representation. In each design in Table II, our algorithm can run PBA for over 1 million crtical paths without any constraints on GPU with 10GB memory.

C. Runtime vs Search Space

In this section, we will demonstrate how the constrained subgraph size, or the *search space*, impacts the runtime of our algorithm on GPU. We run OpenTimer and our algorithm on designs `leon2`, `leon3mp`, `netcard`, `b19_iccad`, `vga_lcd`, `vga_lcd_iccad`, `des_perf_ispd`, and `edit_dist_ispd` under different

path constraints. These path constraints define different constrained subgraphs. We use the number of pins in a constrained subgraph to denote the size. Figure 7 shows the logarithmic runtime of OpenTimer and our algorithm under various sizes of constrained subgraphs. We see that our algorithm is generally faster than OpenTimer at different sizes ranging from $4.3K$ to $2.3M$. We can observe that our algorithm has a clear advantage over OpenTimer on a larger constrained subgraph. For example, in `leon2`, the runtime performance between OpenTimer and our algorithm is quite close at constrained subgraph sizes $5.3K$, $6.7K$, and $7.2K$. However, at size $2.3M$, there is a 4.61 logarithmic runtime difference between our algorithm and OpenTimer, which suggests a $100.5\times$ speed-up. Similar patterns show up in other designs as well. In `leon3mp`, there is only a 2.87 logarithmic runtime difference at size $22K$. At larger sizes $586K$, $588K$, and $594K$, the differences increase to 4.60–4.91, which suggests almost $100\times$ speed-up. In `netcard`, because all constrained subgraphs have sizes below $10K$, the speed-up of our algorithm is minimal. This is expected because, in a highly constrained subgraph, our algorithm does not benefit from much data parallelism for performance improvement.

D. Performance Portability of SYCL Implementation

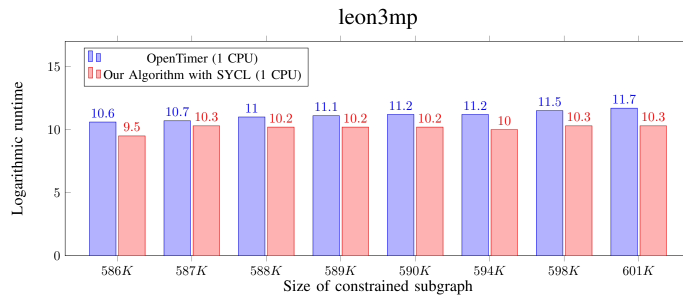


Fig. 8: Logarithmic runtime comparison between OpenTimer and our algorithm on CPU

In this section, we study the performance portability of the SYCL-based implementation. Unlike CUDA, SYCL allows the same C++ code to switch between different SYCL devices in no need of separate codebases. For instance, by changing the SYCL device from GPU to CPU, we can execute our algorithm on CPU with the same SYCL source code. This gives us an insight into how a *data-driven* implementation performs on a CPU architecture that can benefit from modern SIMD/vector parallelism. Figure 8 compares the performance between our SYCL implementation and OpenTimer using one CPU under different constrained subgraphs in `leon3mp`. For constrained subgraph sizes from $586K$ to $601K$, the average logarithmic runtime difference is 0.95; in other words, the same GPU algorithm that runs on CPU has $2.59\times$ overall speed-up over OpenTimer. This result is quite impressive, because it shows that our data-driven algorithm can outperform the state-of-the-art graph-driven algorithm on CPU, when the search space becomes large.

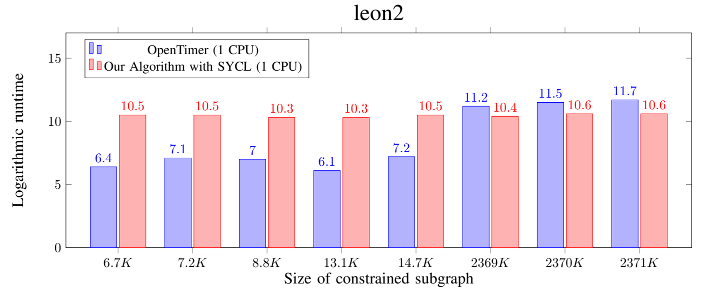


Fig. 9: Logarithmic runtime comparison between OpenTimer and our algorithm on CPU

However, for small search space where data parallelism is limited, this advantage on CPU is not clear. Figure 9 demonstrates this runtime behaviour on benchmark `leon2`. Our `leon2` benchmark contains a mixture of constrained subgraphs, ranging from very small $6.7K$ to very large $2.37M$. We can observe that the size of constrained subgraph affects very little on our SYCL-based implementation on CPU. This is expected because our path filtering strategy is data-oriented. We use array-based filters for constraints satisfaction. The number of iterations for kernel update does not change much for different filters, because we always need data to propagate from endpoint to startpoint. The advantage of this property is that our algorithm's performance is resilient to larger subgraphs, not to mention that we have very high parallelization benefit on GPU. Also, we believe our algorithm can meet practical needs, because optimization flows often request timing report on large constrained subgraph, specially in modern circuit designs. The takeaway here is that our result encourages a new thinking of data-parallel approaches to design critical timing workloads. We believe the modern SYCL programming model can largely facilitate this process given its unique approach to heterogeneous computing using unified single-source C++ programming.

VI. Acknowledgements

The project is supported by the NSF grant CCF-2126672.

VII. Conclusion

To conclude, in this paper, we have introduced a novel GPU-accelerated critical path generation algorithm considering path constraints. We have developed efficient path filters and fast path generation strategy to utilize the throughput advantage of GPU. We have also implemented our algorithm in single-source C++ by using SYCL. It allows us to quickly migrate between different environments. Experiments show that our algorithm achieves $102\times$ speed-up with one GPU on a 1.6M-gate design over the baseline timer. Our algorithm also has high performance portability on large constrained subgraph. Our algorithm can achieve $2.59\times$ speed-up by running entirely on CPU. In future work, we will handle incremental path constraints with a caching framework. Our algorithm will also consider MCMM scenarios by taking advantage of multiple GPUs based on the framework of [19].

References

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

[2] T.-W. Huang and M. D. Wong, "Accelerated path-based timing analysis with mapreduce," in *ACM ISPD*, 2015, p. 103–110.

[3] T.-W. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *ACM/IEEE DAC*, 2016, pp. 116:1–116:6.

[4] T. Huang and M. Wong, "UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR," *IEEE TCAD*, vol. 35, no. 11, pp. 1862–1875, 2016.

[5] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "Fast path-based timing analysis for cppr," in *ICCAD*, 2014, pp. 596–599.

[6] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++," in *IEEE IPDPS*, 2019, pp. 974–983.

[7] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2021.

[8] P. Lee, I. H. Jiang, and T. Chen, "FastPass: Fast timing path search for generalized timing exception handling," in *IEEE/ACM ASPDAC*, 2018, pp. 172–177.

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

[10] K.-M. Lai, T.-W. Huang, and T.-Y. Ho, "A general cache framework for efficient generation of timing critical paths," in *IEEE/ACM DAC*, 2019.

[11] G. Guo, T. W. Huang, C. X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *IEEE/ACM DAC*, 2020, pp. 1–6.

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

[13] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *IEEE/ACM SC*, 2009.

[14] "oneAPI," https://intel.github.io/llvm-docs/GetStartedGuide.html.

[15] "SYCL," https://sycl.tech.

[16] T. Huang and M. Wong, "Opentimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[17] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.

[18] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "Opentimer v2: A parallel incremental timing analysis engine," *IEEE Design and Test*, vol. 38, no. 2, pp. 62–68, 2021.

[19] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020, pp. 1–8.