A High-Performance Heterogeneous Critical Path Analysis Framework

Yasin Zamani
Department of Electrical & Computer Engineering
University of Utah, Salt Lake City, UT

Tsung-Wei Huang
Department of Electrical & Computer Engineering
University of Utah, Salt Lake City, UT

***Abstract*—Emphasis on static timing analysis (STA) has shifted from graph-based analysis (GBA) to path-based analysis (PBA) for reducing unwanted slack pessimism. However, it is extremely time-consuming for a PBA engine to analyze a large set of critical paths. Recent years have seen many parallel PBA applications, but most of them are limited to CPU parallelism and do not scale beyond a few threads. To overcome this challenge, we propose in this paper a high-performance graphics processing unit (GPU)-accelerated PBA framework that efficiently analyzes the timing of a generated critical path set. We represent the path set in three dimensions, timing test, critical path, and pin, to structure the granularity of parallelism scalable to arbitrary problem sizes. In addition, we leverage task-based parallelism to decompose the PBA workload into CPU-GPU dependent tasks where kernel computation and data processing overlap efficiently. Experimental results show that our framework applied to an important PBA application can speed up the state-of-the-art baseline up to 10× on a million-gate design.**

***Index Terms*—Parallel Programming, Static Timing Analysis (STA), Path-Based Analysis (PBA), CUDA Graph**

I. Introduction

Static-timing analysis (STA) is an essential step of the integrated circuit (IC) design flow to verify timing behaviors. STA consists of *graph-based timing analysis* (GBA) and *path-based timing analysis* (PBA). GBA performs a linear scan on the circuit graph and estimates the worst timing quantities at each endpoint. GBA is fast, but the results are pessimistic. Hence, PBA is performed after GBA to reduce unwanted pessimism by reanalyzing the timing with path-specific updates [1]. However, PBA is very time-consuming, typically 10-1000× slower than GBA [2], due to the enormous amount of paths in modern circuits. There are various strategies to reduce the long runtimes of PBA [3]–[6]. However, nearly all of them use central processing unit (CPU) parallelism, and their scalabilities saturate at a few CPU cores. For example, state-of-the-art PBA algorithms that leverage task-based parallelism [4], [5] saturates at 8–10 cores. Figure 1 highlights the runtime challenge of PBA.

To reduce the long runtimes of PBA, we must harness the power of new *high-performance parallelism* comprising manycore CPUs and graphics processing units (GPUs). GPUs have become the primary workhorse behind modern advances in high-performance computing (HPC) applications. Moreover, since HPC is one of the essential tools fueling the advancement of electronic design automation (EDA), therefore, GPU-accelerated EDA tools are one of the fascinating fields to explore. Compared to the CPU, GPU is very powerful in data-intensive applications. This unique performance characteristic of GPU is particularly beneficial for the PBA problem that computes a large number of data paths (e.g., millions of paths in large designs). As a successful example, Guo et al. [7] propose a GPU-accelerated

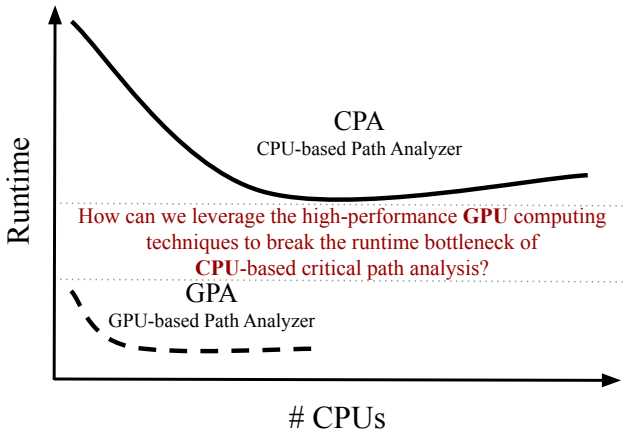


Fig. 1: Runtime of CPU-based path analyzer (CPA) under different number of CPU cores.

algorithm for PBA that shows more than 100× speed-up over a CPU baseline. However, they only target the generation of critical paths and do not (re)analyze the timing path by path, which is another essential step of PBA.

This paper aims to leverage the computation power of GPU to accelerate the path-based timing reanalysis for a generated critical path set. The challenge of this problem is threefold. First, computing critical paths involve very *irregular patterns* because different paths can have different lengths across different timing tests (e.g., hold/setup checks). This irregularity requires strategic decomposition between CPU and GPU to benefit from heterogeneous parallelism. Second, to support a *large number of paths*, we need efficient data structures to fit computations into relatively limited GPU memory. Third, *GPU architectures are very different from CPUs* in thread scheduling, synchronization, and memory hierarchy. This difference requires specially designed algorithms and memory access patterns to achieve high performance.

In this work, we propose a novel CPU-GPU heterogeneous framework to accelerate an essential problem of PBA, reanalyzing the timing of a generated critical path set. We apply our framework to an important PBA application, common path pessimism removal (CPPR). Specifically, we perform a path-specific update to remove unwanted pessimism from the critical path set generated from an updated STA graph and regenerate a timing path report ranked by post-CPPR slacks. We summarize our contributions as follows:

- **General heterogeneous PBA framework**. We propose a three-dimensional representation in terms of tests, critical paths, and pins of path traces. With this cubic representation of the path set, we can structure the granularity of parallelism to arbitrary problem

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

sizes.

- **Task graph-based decomposition**. We leverage the power of the modern CUDA graph to design an efficient parallel decomposition strategy for the PBA workload. Our decomposition strategy transforms the workload into a task dependency graph that flows CPU-GPU dependent operations asynchronously with improved performance.
- **Efficient GPU data structures and kernels**. We design GPU-efficient data structures and algorithm kernels to organize and compute critical paths. The memory complexity of our algorithm is linear to the requested critical path number. Furthermore, we break down our kernels into well-defined algorithm primitives that can compose efficiently on GPUs.

We apply our framework to an important PBA application, CPPR, and evaluate our algorithm on real circuit designs with a golden reference generated by TAU 2014 CPPR Timing Analysis Contest [8], [9]. Our algorithm can analyze millions of critical paths and generate a timing report that matches the golden reference result. Compared to the state-of-the-art PBA engine for CPPR [9], we obtain up to 10× speed-up on million-gate designs using one GPU. At an extreme, our algorithm of one CPU and one GPU is 5–10× faster than the baseline of 16 CPUs. Our algorithm can facilitate PBA to be efficiently integrated into the early design flow to improve the quality of results (QoR) with a reasonable turnaround time. We believe this research opens plenty of opportunities for using high-performance computing techniques to solve STA problems faster.

II. Path-based Timing Analysis

Static timing analysis (STA) computes the propagation of time signals in a circuit from its primary inputs to its primary outputs through various circuit elements and interconnect. STA models the circuit as a directed acyclic graph (DAG) $G = \{V, E\}$. V is the node-set that specifies pins of circuit elements. E is the edge set which specifies pin-to-pin connections. Starting from the primary input(s), we quantify the instant that a signal reaches an input or output of a circuit element as the *arrival time* (at). Similarly, starting from the primary output(s), we quantify the limits imposed for each arrival time to ensure proper circuit operation as the *required arrival time* (rat). Given an arrival time and a required arrival time, we define the *slack* at a circuit node to measure how well timing constraints are met. A positive slack means the required time is satisfied, and a negative slack means the required time is in violation.

Graph-based analysis (GBA) is a step of STA used to perform a worst-case (i.e., early and late) analysis of a circuit over all possible paths to update the graph with timing information, such as parasitics, slew, delay, and arrival time [1]. The early-late timing analysis tries to cover on-chip variations, such as temperature fluctuations and voltage drops, using the worst-case scenarios. The advantage of GBA is its fast, linear-time complexity. However, GBA inherently adds unnecessary pessimism because it only considers the worst-case timing values, which can lead to an over-conservative design [1].

Path-based analysis (PBA) is another step of STA that is typically applied after GBA to reanalyze timing with path-specific updates, such as common path pessimism removal (CPPR), advanced on-chip variation (AOCV), and slew repropagations. PBA can remove unnecessary pessimism from GBA [4]. However, this process is extremely time-consuming because it involves analyzing an exponential number of paths. It has been highlighted that EDA vendors should improve the runtime performance of PBA with new parallel paradigms [10]. Therefore, this paper aims to propose a general PBA framework by harnessing the power of GPU parallelism and applying it to an important PBA application, CPPR.

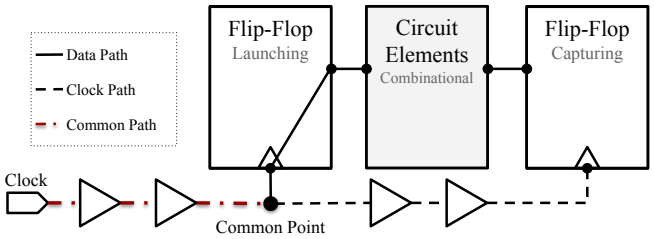


Fig. 2: Clock network pessimism incurs in the common path between the capturing clock path and the launching clock path of a data path.

CPPR removes common clock buffer delays from a data path between its launching path and capturing path. Considering the example in Figure 2, when we perform early-late analysis, we compute the early and the late delays of the common segments between the launching path and the capturing path. However, this computation incurs unnecessary pessimism because the signal cannot simultaneously experience early and late delays. Unnecessary pessimism can lead to tests being marked as failing (having negative slack) when in actuality, they should be passing (having positive slack). CPPR is a particular step in STA to remove unwanted pessimism by introducing a *path-specific* credit, which is the amount of delay of common segments, to add to the path slack (common point (*cp*) is the clock reconverging node of clock-pin of launching and capturing flip-flops):

$$credit = at_{cp}^{late} - at_{cp}^{early},$$

$$slack_{post\_CPPR} = slack_{pre\_CPPR} + credit.$$

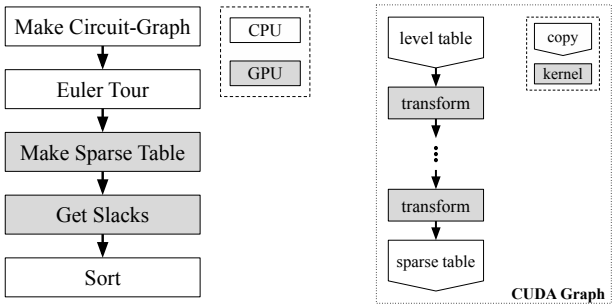
Problem formulation: *Given a set of generated critical paths from an updated STA graph after GBA, rank these paths based on the increasing order of their post-CPPR slacks.*

While this paper concentrates on an important PBA application, CPPR, we aim to propose several *framework-neutral* GPU algorithms and parallel decomposition strategies that are generally applicable to other PBA problems. Our emphasis is on reanalyzing the timing of generated critical paths with path-specific updates rather than tailoring GPU algorithms to a specific PBA problem. This architectural decision makes our work general and complements the state-of-the-art GPU-accelerated path generator [7] which can be our input.

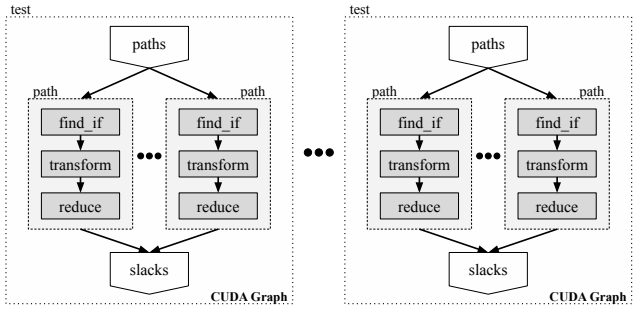
III. GPU-accelerated Path Analysis Framework

Figure 3 shows the overview of our GPU-accelerated PBA framework with a specific focus on CPPR application. The gray blocks and the white blocks denote the computation on GPU and CPU, respectively. On the GPU side, pentagons show the copying data between host and device, and rectangles show compute kernels. We start by constructing a circuit graph from given delay and timing data files. Then, we build some look-up tables via Euler tour starting at the root of the clock tree. Then, on the GPU side, we make the sparse table. After that, We group different paths in different timing tests and transfer them to the GPU memory. Analyzing critical paths on the GPU side consists of three main primitive steps: (1) *find_if*. (2) *transform*. (3) *reduce*. In the end, for generating the timing report, we sort the timing tests and containing critical paths based on computed post-CPPR slacks.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE



(a) Analyze Paths (b) Make Sparse Table



(c) Get Slacks

Fig. 3: Overview of our GPU-accelerated PBA algorithm. We leverage the power of the modern CUDA graph to decompose the PBA workload into a task dependency graph that offloads dependent GPU operations asynchronously with low kernel launch overheads. Furthermore, we break down our GPU kernels into general algorithm primitives, such as *find_if*, *transform* and *reduce*, that can be executed efficiently with existing highly optimized CUDA toolchains.

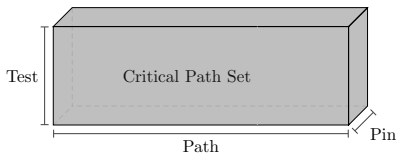


Fig. 4: Parallel decomposition of our path analysis framework in terms of tests, critical paths, and pins.

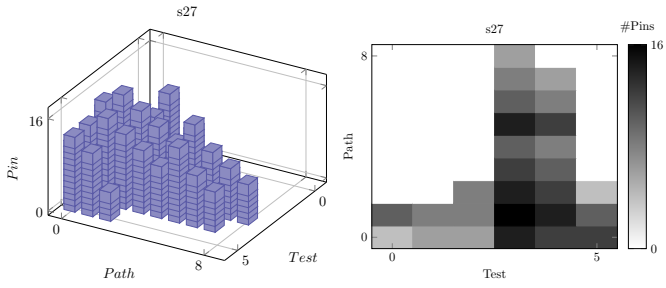


Fig. 5: Uneven distribution of paths among tests causes irregular parallelism on GPU. s27 is a benchmark from TAU 2014 CAD Contest [8]

A. GPU Data Structures for Circuit Graph and Critical Paths

To offload PBA to GPU, we must efficiently represent the circuit graph on GPU. We collect all fan-out or outgoing edges of each vertex is denoted as graph G. Because G is a sparse graph, we use the Compressed Sparse Row (CSR) format to represent it. CSR is one of the most common graph formats used in GPU applications [11], [12]. CSR requires three one-dimensional arrays to represent a weighted directed graph. The format includes a vertex array for row offsets, an edge array for column values, and a weight array for weights of all edges. Therefore, CSR is highly memory efficient. The total size of CSR is only $N + 2M$ for a graph with vertex number N and edge number M.

In addition to the STA graph, we need to transfer the critical path set data to the GPU memory. We propose a cubic model to represent the critical paths. As Figure 4 shows, the set of critical paths is represented in terms of timing-test, critical path, and pin of path traces. This model can help us to structure the granularity of parallelism to arbitrary problem sizes. Another important thing, in addition to a large number of critical paths, is the very different lengths of paths. Figure 5 shows this issue for the simple circuits s27 [8]. We know that subgroups of different paths usually have the same properties that can be shared for less execution time and more efficient use of memory (for example, a subset of paths may have the same required arrival time, or they may all have the same sub-path). Therefore, we separate paths into different groups based on these similarities and then dispatch each group into thousands of GPU threads. Our strategy scales to millions of paths during the analyzing process.

This step aims to leverage the power of the CUDA graph to design an efficient parallel decomposition strategy for the PBA workload that can transform the workload into a task dependency graph that flows GPU-dependent operations (e.g., kernel or memory copy) asynchronously with improved performance. Figure 6 shows the part of the CUDA graph for benchmark s27. CUDA has recently introduced a new graph programming model, namely CUDA graph [13], that allows users to describe a large GPU workload in a single task graph and offload the task graph directly onto a GPU using a single CPU call. The new CUDA graph execution model has demonstrated significant success. For instance, the recent research at 2021 Nvidia GTC has shown over 3× performance improvement in TensorFlow by replacing stream-based execution with CUDA graph [14].

Because the number of critical paths is extremely large and we need to call several kernels to process each path, it spends a significant amount of time on the CPU to offload kernels to the GPU. Therefore, by launching multiple GPU operations through a single CPU call, we can overcome the overheads of many kernel calls. For this reason, we first categorize critical paths into corresponding timing tests, then put several tests in groups. By describing a GPU workload in a task dependency graph, rather than aggregated GPU operations and dependencies, we allow the runtime to run whole-graph scheduling optimization to improve performance significantly.

B. GPU Kernels for Critical Path Analysis

Algorithm 1 shows the overall path analysis function. It consists of two main stages in the GPU side: *making sparse table* and *getting pessimism-free slacks*. The first stage aims to tabulate the common path information for a quick lookup of credit, while the goal in the second stage is calculating the post-CPPR slacks to identify the top-k critical paths from a given test. The generic framework of our project is developed based on analyzing timing tests (i.e., hold and setup tests) independently. In other words, each test is treated as an

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

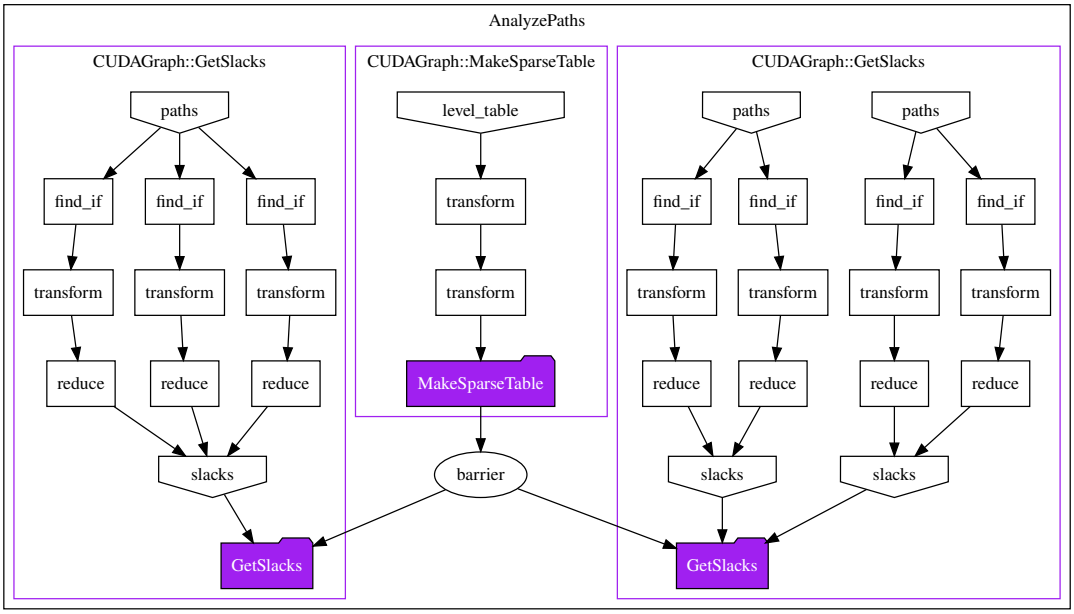


Fig. 6: Partial CUDA graph for s27 benchmark. We use the CUDA graph to design an efficient parallel decomposition strategy for the PBA workload. We transform the workload into a task dependency graph that can flexibly extend to accommodate different groups of tests. Each CUDA graph of many GPU-dependent operations (e.g., kernel, memory copy) will be submitted to the CUDA runtime using a single CPU call to reduce the kernel call overheads. The CUDA runtime will schedule these GPU operations asynchronously.

independent input without dependency on the others. For structuring the granularity of parallelism, a readily available parallel extension can be carried out by evoking multiple threads with each operating on each test. With the shared lookup table and the circuit graph, we impose the low memory requirement by maintaining only private information for each thread. We can simultaneously send several tests to the maximum number of CPU cores supported by the machine can to the GPU side for path analysis.

Algorithm 1 ANALYZEPATHS

Input: critical paths grouped in $tests$
Output: sort $paths$ in each $test$ based on post-CPPR $slacks$

```
1: delay, at ← MAKECIRCUITGRAPH
2: E, L, H ← EULERTOUR
3: M ← MAKESPARSETABLE(L)
4: slacks ← GETSLACKS(tests, delay, at, E, L, H, M)
5: comp ← (a, b) ⇒ slacks[a] < slacks[b]          ▷ comparison
6: for all test ∈ tests do
7:     SORT(paths ∈ test, comp)
```

1) Make Saprse Table on GPU: The goal of this step is to make a sparse table on GPU. With the shared sparse table, we impose the least memory requirement by maintaining only private information about the critical paths for each thread. By creating the sparse table on the GPU side, we save time transferring sparse table data from host to device. Since each row in the sparse table depends only on the previous row, we can use the transform function to construct each row according to the information in the previous row.

In graph theory, the clock reconverging node of two nodes in the clock tree is equivalent to the two nodes' lowest common ancestor (LCA). The LCA of two nodes u and v in a tree is the lowest (i.e., deepest) node with both u and v as descendants, where we define each node to be a descendant of itself. The arrival time information of each node in the clock tree can be precomputed, and therefore the credit of two nodes can be obtained immediately once their LCA is known. Many state-of-the-art LCA algorithms have been invented over the last decades. The table-lookup algorithm by [5] is employed as our LCA engine due to its simplicity and efficiency. Since circuit graph values will not change, the sparse table is allowing $O(1)$ query answering with $O(N \log N)$ build time [9]. For a given clock tree, we build four tables as follows:

- The Euler table E records the identifiers of nodes in the Euler tour of the clock tree; $E[i]$ is the identifier of ith visited node.
- The level table L records the levels of nodes visited in the Euler tour; $L[i]$ is the level of node $E[i]$.
- The occurrence table $H[v]$ records the index of the first occurrence of node v in the array E.
- Two-dimensional table $M[i][j]$ stores the index of the minimum value in the level table starting at i having length 2^i [15].

Tables E, L, and H can be built using depth-first search starting at the root of the clock tree [16]. The procedure of building these tables is done before calling the Algorithm 2. Algorithm 2, fullfiles the table M via bottom-up dynamic programming with the follow [9]:

$$M[i][j] = \begin{cases} i, & j = 0 \\ M[i][j-1], & L[M[i][j-1]] \leq \\ & L[M[i+2^{j-1}][j-1]] \\ M[i+2^{j-1}][j-1], & o.w. \end{cases}$$

Provided the table M has been processed, the value of $min_L(a, b)$ can be computed by selecting two blocks that entirely cover the interval between a and b and returning the minimum between them. Let c be $\lfloor \log(b - a + 1) \rfloor$ and assume $b > a$, the following formula

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

is used for computing the value of min_L:

$$min_L(a,b) = \begin{cases} M[a][c], & L[M[a][c]] \leq \\ & L[M[b-2^c+1][c]] \\ M[b-2^c+1][c], & o.w. \end{cases}$$

As a result, the LCA of a node pair (u, v) is the node situated on the smallest level between the first occurrence of v and the first occurrence of u. Denoting the index of the node with the smallest level between the index a and b in the level table L as $min_L(a, b)$, the LCA of a given node pair (u, v) is

$$lca(u,v) = E[min_L(H[u], h[v])].$$

Algorithm 2 MAKESPARSETABLE

Input: level table L
Output: sparse table M

```
1:  n ← |L|                                          ▷ number of columns
2:  m ← ⌈log(n)⌉                                     ▷ number of rows
3:  M ← GPU::MALLOC(n × m)
4:  first ← address of M[0][0]
5:  last ← address of M[0][n]
6:  GPU::IOTA(first, last, 0)                        ▷ first row (i = 0)
7:  for i ← 1 to m do                                ▷ other rows (i > 0)
8:      first_1 ← address of M[i − 1][0]
9:      last_1 ← address of M[i − 1][n − 2^(i−1)]
10:     first_2 ← address of M[i − 1][2^(i−1)]
11:     output ← address of M[i][0]
12:     op ← (a, b) ⇒ L[a] < L[b] ? a : b            ▷ binary operation
13:     GPU::TRANSFORM(first_1, last_1, first_2, output, op)
```

2) Get Slacks on GPU: This step aims to update slacks of paths in parallel by removing common path pessimism using GPU. Updating slacks is the most time-consuming step due to a large number of critical paths. Using algorithms find-if, transform, and reduce as primitive, we can identify the top-k critical paths. We first find the clock pin of the launching flip-flop using the find_if function for a given path. Since we know the arrival time of the founded pin, we consider only the part from this pin to the end of the path (data pin of capturing flip-flop) to continue the analysis. Using the transform function, we map both consecutive pins to arc delay between them. Finally, using the reduce function, we add all the delays mapped from the previous step and calculate the delay of the path.

Algorithm 3 calculates post-CPPR slacks for all paths of a given test. The outermost for loop starts at line 1, is developed based on one test at one time. In other words, each test is treated as an independent input without dependency on the others. We can simultaneously transfer data of several tests with up to the maximum number of CPU cores supported by the machine to the GPU side for path analysis. The inner loop starts at line 2, is a parallel extension to operating on each path of the selected test. Line 6 finds the clock pin of launching flip-flop. Line 13 calculates the pre-CPPR slacks of the given path. Using tables E, L, H, and M as infrastructure, we can retrieve Line 16 calculates the credit of two given nodes in the clock tree in constant time.

IV. EXPERIMENTAL RESULTS

This section demonstrates that our GPU-Accelerated algorithm can generate an accurate path report (tests/paths must be sorted based on post CPPR slacks) on numerous critical paths faster than the CPU path analyzer. We performed our experiments on a 64-bit CentOS

Algorithm 3 GETSLACKS

Input: critical paths grouped in $tests$
Input: arc delays $delay$
Input: clock-tree arrival times at
Input: Euler table E, level table L, occurrence table H, and sparse table M to calculate lca
Output: post-CPPR $slacks$

```
1:  for all test ∈ tests do                          ▷ parallel on CPU
2:      for all path ∈ test do                       ▷ parallel on GPU
3:          first ← address to the beginning of path
4:          last ← address to the end of path
5:          p ← (pin) ⇒ pin ∈ clock-tree             ▷ unary predicate
6:          src ← GPU::FIND_IF(first, last, p)
7:          next ← address to the successor of src
8:          output ← address of delay
9:          op ← (u, v) ⇒ delay_{u→v}                ▷ binary operation
10:         GPU::TRANSFORM(src, last, next, output, op)
11:         first ← address to the beginning of delay
12:         last ← address to the end of delay
13:         slacks[path] ← GPU::REDUCE(first, last, at_src)
14:         tgt ← clock-pin of capturing flip-flop
15:         cp ← lca(src, tgt)                       ▷ common point
16:         credit ← at_cp^late − at_cp^early
17:         slacks[path] ← slacks[path] + credit
```

Stream 8 Linux machine with one NVIDIA GeForce RTX 3060 GPU (Compute Capability 8.6) and eight 2.90GHz Intel Core i7 CPU cores. We compiled our programs with Nvidia CUDA NVCC 11.1 device compiler and GNU GCC 9.2.1 host compiler, where optimization flag `-O3` and C++17 standard `-std=c++17` are enabled. We used 1024 threads per block for all kernel configurations and used Taskflow [17] for our task graph programming. We considered the top-ranked UI-Timer from TAU 2014 CAD Contest [9] as our CPU path analyzer baseline. We evaluated our algorithm on the largest TAU 2014 Timing Contest benchmarks [8] which are described in Table I.

A. Path Report Accuracy

Our algorithm can generate an accurate timing report. In order to evaluate its accuracy, we used our algorithm and UI-Timer to generate separate timing reports with the same number of critical tests/paths. We took the absolute slack difference between every pair of critical tests/paths in two reports and recorded the maximum difference value. We performed this evaluation with up to a million critical paths (1024 tests and 1024 paths for each test) on the eight largest TAU 2014 Timing Contest benchmarks. The statistics of each benchmark are shown in Table I. Our algorithm reports identical critical paths as UI-Timer.

B. Runtime Performance

Our algorithm can accelerate finding top critical tests/paths. We compared the runtime of our algorithm (1 GPU) with the runtime of the UI-Timer as CPU path-based analyzer by reporting top critical paths on the eight largest benchmarks in TAU 2014 Timing Contest [8]. Details of our experimental results are shown in Table I. We can observe that our algorithm achieves speed-up on million-gate designs over UI-Timer. We speed-up the baseline 6.7× on vga_lcd (0.5M gates), 6.1× on Combo7 (2.8M gates), and 9.7× on Combo6 (3.6M gates). Our algorithm also achieves speed-up on medium benchmarks, such as 1.9× on des_perf, 2.5× on Combo4.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

TABLE I: Elapsed time (seconds) comparison between CPA and GPA. Benchmarks are from TAU 2014 CAD contest [8].

Benchmark	|**V**|	|**E**|	|**C**|	**# Tests**	**# Paths**	**CPA** (1 CPU)	**GPA** (1 CPU, 1 GPU)		**CPA** (8 CPUs)	**GPA** (8 CPUs, 1 GPU)	
des_perf	330538	404257	88751	19764	1682	14.130	**7.550**	**(1.9x)**	10.402	**7.524**	**(1.4x)**
vga_lcd	449651	525615	172065	50182	5281	35.738	**5.368**	**(6.7x)**	30.250	**5.312**	**(5.7x)**
Combo2	260636	284091	171529	29574	62938	21.288	20.525	(1.0x)	13.951	20.235	(0.7x)
Combo3	181831	284091	73784	8294	129854	10.606	16.067	(0.7x)	7.892	16.077	(0.5x)
Combo4	778638	866099	469516	53520	19227963	76.661	**30.264**	**(2.5x)**	70.771	**29.543**	**(2.4x)**
Combo5	2051804	2228611	1456195	79050	19227963	293.538	**58.676**	**(5.0x)**	270.182	**58.658**	**(4.6x)**
Combo6	3577926	3843033	2659426	128266	19227963	786.822	**81.402**	**(9.7x)**	729.348	**80.934**	**(9.0x)**
Combo7	2817561	3011233	2136913	109568	19227963	532.569	**87.558**	**(6.1x)**	501.214	**87.519**	**(5.7x)**

|**V**|: size of node set. |**E**|: size of edge set. |**C**|: size of clock tree. **#Tests**: number of hold/setup tests. **#Paths**: max number of data paths per test.

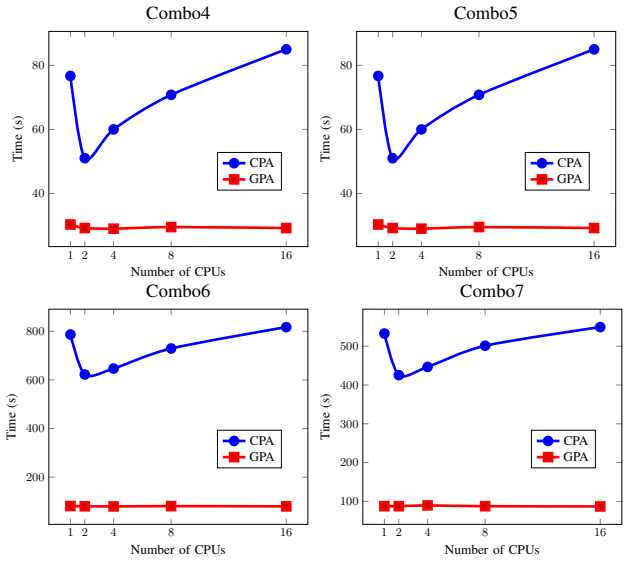


Fig. 7: Runtime comparison of CPA and GPA under a different numbers of CPUs for the four largest benchmarks.

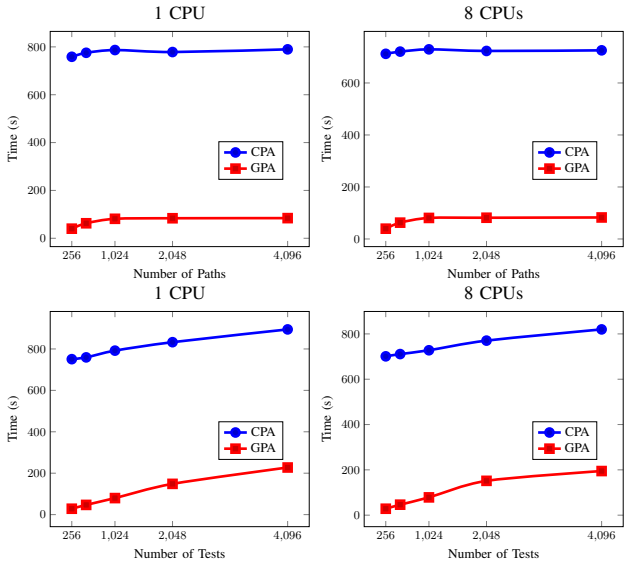


Fig. 8: Runtime comparison of CPA and GPA under a different number of Path/Tests for the largest benchmark, Combo6.

To demonstrate our performance advantage over the baseline, Figure 7 plots the speed-up curve of our algorithm over the baseline across different numbers of CPU cores. We observe that baseline saturates' performance improves as the number of cores increases, and there is always a significant performance margin to ours. With the baseline at any CPU concurrency of multiple cores, our algorithms are still faster than baseline on the four largest designs Combo4, Combo5, Combo6, and Combo7. In fact, according to our experiments, our GPU-accelerated PBA algorithm is always faster than baseline in all designs larger than vga_lcd (Table I shows that for small circuits such as Combo2 and Combo3, the CPA is faster than the GPA), regardless of the number of CPU cores the baseline used.

Finally, we investigate the scalability of our GPU-accelerated PBA algorithm by varying the input parameter of the path count from 256 to 4096 for 1024 tests and test count from 256 to 4096 for the 1024 paths. The performance comparing GPA with CPA on the largest circuit, Combo6, is characterized in Figure 8. We see that all runs are accomplished by our GPA algorithm faster than CPA, and the runtime gap is evident. To sum up, in precise, these results have justified the practical viability of our GPA algorithm.

V. Acknowledgment

This work is supported by an NSF Grant CCF-2126672 and NumFOCUS Small Development Grant under the Taskflow project.

VI. Conclusion

This paper has introduced a novel general CPU-GPU heterogeneous PBA framework to overcome the runtime bottleneck of CPU-based PBA. We decompose the critical path analysis into multiple GPU-accelerated primitive kernels. Moreover, we leverage the cubic path representation method to design GPU-efficient data structures and structure parallelism's granularity scalable to arbitrary problem sizes. We apply our framework to an important PBA application, the CPPR problem. Experiments show that our algorithm achieves up to 10× speed-up on million gate designs over the state-of-the-art PBA algorithms. Our algorithm can promote PBA in the earlier stage of design closure flow to improve QoR and turnaround time. We believe this research opens plenty of opportunities for using high-performance computing techniques to solve STA problems faster and will inspire other workloads that share similar performance characteristics.

Our future work plans to extend our PBA algorithm to multiple GPUs and implement a GPU-efficient indirect sorting algorithm to work well on uneven distribution paths. Furthermore, we plan to renovate the GPU-parallel infrastructure of existing heterogeneous STA algorithms [7], [18] using GPU task graph parallelism and programming models [19].

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

REFERENCES

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

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

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

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

[6] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "opentimer v2: A new parallel incremental timing analysis engine," *IEEE TCAD*.

[7] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1–6.

[8] "Tau 2014 contest: Pessimism removal of timing analysis," http://sites.google.com/site/taucontest2014.

[9] T.-W. Huang, P.-C. Wu, and M. D. Wong, "Ui-timer: An ultra-fast clock network pessimism removal algorithm," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2014, pp. 758–765.

[10] "EDA Vendors Should Improve The Runtime of PBA," https://www.electronicdesign.com/technologies/eda/article/21796368.

[11] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[12] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *IEEE/ACM SC*, 2009, pp. 1–11.

[13] "Nvidia cuda graph," https://developer.nvidia.com/blog/cuda-graphs/.

[14] A. A. Awan, J. Bédorf, C.-H. Chu, H. Subramoni, and D. K. Panda, "Scalable distributed dnn training using tensorflow and cuda-aware mpi: Characterization, designs, and performance evaluation," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019, pp. 498–507.

[15] M. A. Bender and M. Farach-Colton, "The lca problem revisited," in *Latin American Symposium on Theoretical Informatics*. Springer, 2000, pp. 88–94.

[16] R. E. Tarjan and U. Vishkin, "Finding biconnected componemts and computing tree functions in logarithmic parallel time," in *25th Annual Symposium onFoundations of Computer Science, 1984.* IEEE, 1984, pp. 12–20.

[17] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2021.

[18] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *2020 ACM/IEEE International Conference on Computer-aided Design (ICCAD)*, 2020.

[19] D.-L. Lin and T.-W. Huang, "Efficient GPU Computation using Task Graph Parallelism," in *2021 European Conference on Parallel and Distributed Computing (Euro-Par)*.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE