GPU-accelerated Path-based Timing Analysis

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 an important step in the design closure flow for reducing slack pessimism. However, PBA is extremely time-consuming. Recent years have seen many parallel PBA algorithms, but most of them are architecturally constrained by the CPU parallelism and do not scale beyond a few threads. To overcome this challenge, we propose in this paper a new fast and accurate PBA algorithm by harnessing the power of graphics processing unit (GPU). We introduce GPU-efficient data structures, high-performance kernels, and efficient CPU-GPU task decomposition strateiges, to accelerate PBA to a new performance milestone. Experimental results show that our method can speed up the state-of-the-art algorithm by 543× on a design of 1.6 million gates with exact accuracy. At the extreme, our method of 1 CPU and 1 GPU outperforms the state-of-the-art algorithm of 40 CPUs by 25–45×.**

I. Introduction

Path-based Analysis (PBA) is pivotal for achieving accurate timing results by reducing unwanted pessimism in Static Timing Analysis (STA) [1]. However, PBA is extremely time-consuming, typically 10-1000× slower than graph-based analysis (GBA) [2]. The high runtime cost has imposed a significant barrier for designers to incorporate PBA in the early design closure flow to improve Quality of Results (QoR) in the timing landscape. To alleviate the long runtime of PBA, existing works have proposed various strategies [3], [4], [5], [6], [7], [8]. However, nearly all of them are architecturally constrained by CPU parallelism, and their results stagnate at a few CPU cores. For example, the state-of-the-art PBA algorithm [3], [4] adopts task-based parallelism with exact accuracy, but its performance saturates at 16 cores. Jin [6] proposes a fast and accurate block-based algorithm that improves runtime up to 1.63×, but proposed algorithm is highly sequential and scales to fewer CPU cores. Peng [7] modifies GBA and presents a path-oriented calculation model, but its improvement on accuracy is limited. Guo [9] has used GPUs to accelerate RC updates and graph levelization in GBA. Yet the same idea cannot be applied to PBA because PBA requires path-specific management and analysis.

As illustrated in Figure 1, fundamental computational challenges of PBA remain unsolved. Current STA engines have very limited performance gain by counting on multi-core CPUs. To achieve transformational performance milestone, new PBA algorithm must harness the power of heterogeneous parallelism, *CPU-GPU hybrid computing*. Nevertheless, offloading PBA to GPU is an extremely challenging task for three reasons. Firstly, PBA is graph-oriented and involves *irregular* computational patterns, requiring very strategic decomposition between CPU and GPU to benefit from heterogeneous parallelism. Secondly, the dynamic process of path generation needs *specially-designed* GPU kernels to search for critical path and maintain path priorities. Lastly, to support a large number of paths, we need efficient data structures to overcome the hurdle of relatively limited GPU memory.

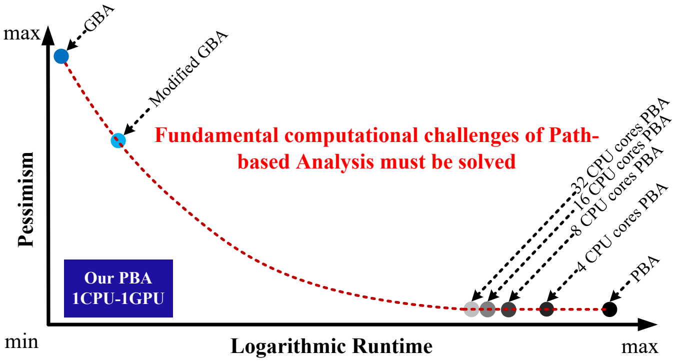


Fig. 1: Computational trade-off between runtime and pessimism reduction on path-based timing analysis.

In this work, we propose a novel GPU-accelerated PBA algorithm to overcome the challenges of long runtimes of PBA. We focus on the core building block, *critical path generation*, which spends the most time of PBA [1]. Specifically, we identify a set of paths in a decreasing order of their criticality from an updated STA graph such that any PBA algorithms or frameworks can perform path-specific update on the path set. We highlight three key contributions as follows:

- **GPU-accelerated path search algorithm**. We design a GPU-accelerated algorithm that iteratively explores critical path candidates from the previously identified critical paths. We construct a shortest path forest by unifying the worst critical paths at different flip-flops and primary outputs with GPU threads. Starting with the forest, we explore new critical path candidates by alternating path prefixes concurrently.
- **GPU-efficient data structures**. We organize critical path candidates in a specially designed array structure that is highly efficient for GPU. The total memory complexity is linear to the circuit graph size and the requested critical path count. During the kernel execution, each set of newly explored path candidates are dynamically allocated in an

efficient one-dimensional (1D) array. Based on the priorities of critical paths (i.e., slack), we prune and compress the path set to improve GPU memory efficiency.

- **Scalable to large numbers of critical paths**. Our path search algorithm enables each GPU thread to independently explore new critical path candidates without contending with other threads. We separate critical path candidates into different groups based on their path prefixes and we dispatch each group into thousands of GPU threads. Our strategy scales to millions of paths during the search process.

We evaluate our algorithm on real designs with a golden reference generated by an industrial standard timer [10]. Our algorithm can scale to millions of critical paths that match the result of the golden reference. Compared to the state-of-the-art path generation algorithm [3], [8], we obtain up to 543× speed-up on million-gate designs. At the extreme, our algorithm of 1 CPU and 1 GPU is 25–45× faster than the baseline of 40 CPUs. Our algorithm can enable designers to incorporate PBA earlier in the design flow to imporve QoR with reasonable turnaround time.

II. Path-based Timing Analysis

PBA is a pivotal step in STA [1]. STA models the circuit as a *directed acyclic graph* (DAG). Each vertex in the graph represents a pin and each edge represents a pin-to-pin connection. A typical STA flow performs GBA first to update the graph with timing information, such as parasitics, slew, delay, and arrival time, at the worst-case scenario (i.e., min and max) [1]. GBA is fast but pessimistic. Therefore, PBA is performed after GBA to update the slack with path-specific properties, including common path pessimism removal (CPPR), advanced on-chip variation (AOCV), and so on to remove unwanted pessimism [3]. Among various PBA frameworks, identifying a set of critical paths to analyze is imperative. However, this process is extremely time-consuming and involves computing an exponential number of paths that can take several hours to finish. It has been highlighted that EDA vendors should improve the runtime performance of PBA with new parallel paradigms [11].

III. Proposed GPU-accelerated PBA

Figure 2 shows the overview of our GPU-accelerated PBA algorithm. The blue block and the white block denote the computation on GPU and CPU, respectively. We start off by constructing a shortest path forest based on a updated STA graph. Then, we iteratively explore critical path candidates by permuting path prefixes. Each iteration consists of three heterogeneous steps: (1) *Look-ahead Level Allocation*. (2) *Interlevel Expansion*. (3) *Intralevel Compression*. We define the set of path candidates with the same number of path prefix permutations as a *level set*. We maintain a level counter to record the number of expanded levels. When the level counter reaches a threshold of decent accuracy (tunable depending on the GPU capability), we stop the iteration and derive the final critical paths from the implicit path representation on GPU.

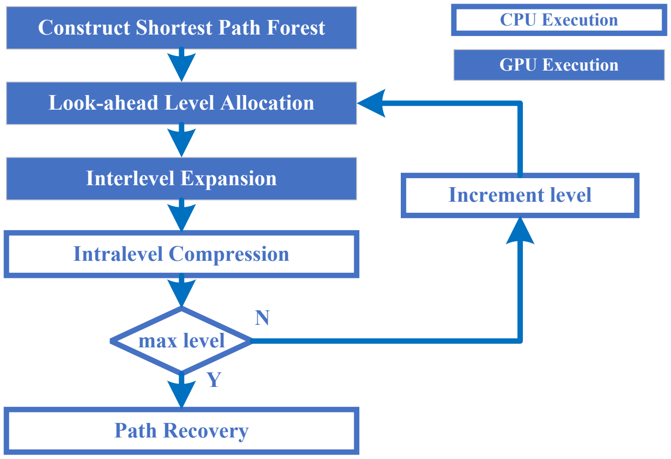


Fig. 2: Overview of our GPU-accelerated PBA algorithm.

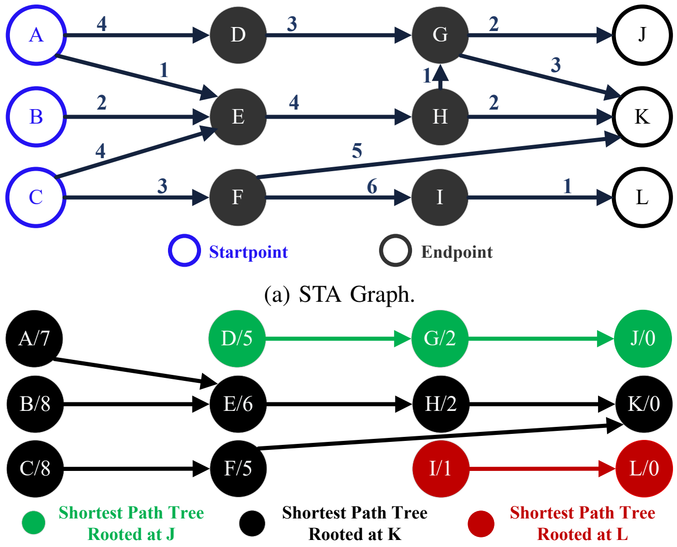
A. STA Graph Structure on GPU

To offload PBA to GPU, we must efficiently represent the STA graph on GPU. We collect all fan-in or incoming edges of each vertex and denote this collection as graph G^-. Similarly, the collection of all fan-out or outgoing edges of each vertex is denoted as graph G^+. We use the Compressed Sparse Row (CSR) format to represent G^- and G^+. CSR is one of the most common graph formats used in GPU applications [12]. 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, 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.

B. Shortest Path Forest

To efficiently enumerate paths on GPU, we introduce a compact data structure, *shortest path forest*, to hold path suffix information. Existing methods [3], [8] construct an independent shortest path tree at each datapath endpoint regardless of overlap between trees. This causes redundant tree construction in overlapped area and waste of memory. Instead, we unify all shortest path trees into a forest that includes a predecessor array `forest[N]` and a distance array `distances[N]`, which eliminates redundant construction at the same vertex. Figure 3 illustrates a *shortest path forest* (3b) constructed from the STA graph (3a). The shortest path forest consists of three shortest path trees rooted at vertices J, K, and L, respectively. We can observe that trees rooted at J and L are not fully built to the startpoints, because the tree rooted at K has smaller cumulative distances at overlapped vertices. This saves the effort to build the trees J and L all the way to vertex A and C. By merging shortest path trees together, our *shortest path forest* is much more compact and memory-efficient than building each independent shortest path tree.

We leverage GPU to construct the shortest path forest. Inspired by the GPU-accelerated Dijkstra's algorithm [13], [14], we mark all datapath endpoints as destinations and


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

(a) STA Graph.

(b) Shortest path forest.

Fig. 3: Shortest path forest generation on GPU.

propagate the shortest distances concurrently with required arrival time as their initial offsets. Our algorithm iteratively invokes the distance update kernel in Algorithm 1 to propagate shortest distances until no distances can be updated. Algorithm 1 assigns each thread a vertex (line 1 and 2). If the assigned vertex has been updated (line 5), its thread needs to propagate distances to adjacent vertices (line 15). The distance array `distances[N]` will be finalized after convergence. We can recover the edge array or predecessor array `forest[N]` by matching edge weight with the distance array. Algorithm 1 can quickly construct the shortest path forest because the GPU kernel updates a batch of vertices concurrently at each iteration.

C. Look-ahead Level Allocation

The goal of look-ahead level allocation is to arrange the output locations of each thread prior to expansion so that thread contention can be avoided. We take the idea of row offset in the CSR graph format into our critical path data structure. The output location of each critical path in the expansion is strictly tied to parent of the path. We compute and sum over the number of critical path candidates that originate from each previously identified critical path. We leverage the implicit critical path representation in the state-of-the-art algorithm [3] for memory efficiency. Given the *shortest path forest* constructed in Section III-B, we separate edge set of the STA graph into two groups. The group of edges in the *shortest path forest* are defined as *suffix edges*. The rest of edges are defined as *deviation edges*. We can represent any critical paths with *deviation edges* because the explicit path trace can be recovered later by complementing with *suffix edges*. The number of *deviation edges* in this implicit representation is denoted as *deviation level*. All critical path candidates with the same *deviation level* are saved in a compact 1D array. For instance, the set of shortest paths involves no *deviation edges* and these paths compose *deviation level 0*.

Algorithm 1: Propagate Distance Kernel

```
   Input  : $G^-$ in CSR format, $N$ as #vertices, $M$ as #edges,
            vertices[$N$], edges[$M$], weights[$M$]
   Input  : Shortest distance cache, distanceCache[$N$]
   Input  : Array indicating vertices with updated distances,
            distanceUpdated[$N$]
   Result: Shortest distances array, distances[$N$]
1  tid ← blockIdx.x * blockDim.x + threadIdx.x;
2  if tid ≥ N then
3      return;
4  end
5  if distanceUpdated[tid] is false then
6      return;
7  end
8  distanceUpdated[tid] ← false;
9  edgeStart ← vertices[tid];
10 edgeEnd ← (tid == N-1) ? M : vertices[tid+1];
11 for eid ← edgeStart to edgeEnd do
12     neighbor ← edges[eid];
13     weight ← weights[eid];
14     newDis ← distances[tid] + weight;
15     atomicMin (&distanceCache[neighbor], newDis);
16 end
17 return;
```

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`	path slack value

Data fields for each *deviation edge* are shown in Table I. We use `parent` and `childOffset` to establish level connections. Data field `parent` can backtrace the critical path parent until one of the shortest paths is reached. Data field `childOffset` can keep track of the output locations of newly explored path candidates.

In the first kernel of this step, shown in Algorithm 2, we assign each *deviation edge* entry in current level with one GPU thread (line 1 and 5). The task of each thread is to compute the number of child deviations (line 7) and save the number in `childOffset` (line 8). We then launch prefix-sum kernels to obtain the correct offset. The `childOffset` of the last entry in current level will be the total number of explored paths in the next level. We can use this number for dynamic allocation of next level. Up to this point, preparation for level expansion is complete.

D. Interlevel Expansion

The objective of this step is to write all data fields of *deviation edge* except `childOffset` in the new level. Figure 4 illustrates expansion process from *deviation level 0* to

```
Algorithm 2: Compute Path Count Kernel
Input  : $G^+$ in CSR format, $N$ as #vertices, $M$ as #edges,
         vertices[$N$], edges[$M$]
Input  : Shortest path forest edge array, forest[$N$]
Input  : currLevel as current level
Input  : levelSize as the number of entries in current level
Result: Compute path numbers originated from current level
tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
if tid ≥ levelSize then
    return;
end
v ← currLevel[tid].to;
/* Get the number of deviation paths
   starting from vertex v in the forest */
pathNum ← getPathNum(v, vertices, edges, forest);
currLevel[tid].childOffset ← pathNum;
return;
```

deviation level 1 on the STA graph from Figure 3a. The expansion begins with *deviation level 0*, $\{e_{\emptyset A}, e_{\emptyset B}, e_{\emptyset C}\}$, which represents shortest paths set $\{AEHK, BEHK, CFK\}$. As for notation, $e_{\emptyset A}$ is a virtual *deviation edge* that means the shortest path starting from vertex A.

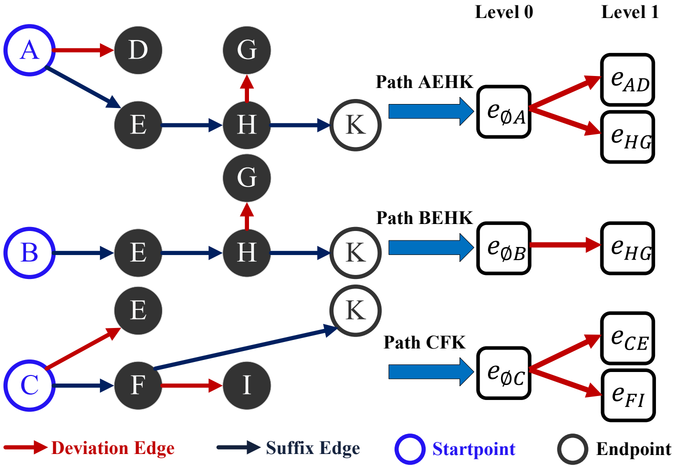


Fig. 4: GPU expansion of the first level.

When we expand next level, we traverse each critical path P of current level and look for *deviation edges* along the path. These *deviation edges* will be filled out in the next level as children of P. For example, shortest path from vertex A, or $e_{\emptyset A}$ in implicit representation, connects to two *deviation edges* e_{AD} and e_{HG}. In *deviation level 1*, $\{e_{AD}, e_{HG}\}$ or paths $\{ADGJ, AEHGJ\}$ are children of $e_{\emptyset A}$. Same procedure happens simultaneously for shortest paths from B and C. For a *deviation edge* e_{vu} along the path P, the slack value $slack_{new}$ of this new path can be computed with the following formula:

$$slack_{new} = slack_P + distance(u) + weight(e_{vu}) - distance(v)$$

Algorithm 3 shows the kernel implementation that complies with the expansion rules above. We launch the kernel with one *deviation edge* in current level per GPU thread (line 1). Each GPU thread is tasked to fill out all the children information in

```
Algorithm 3: Expand New Level Kernel
Input  : $G^+$ in CSR format, $N$ as #vertices, $M$ as #edges,
         vertices[$N$], edges[$M$], weights[$M$]
Input  : Shortest path forest, forest[$N$] as edge array,
         distances[$N$] as distance array
Input  : currLevel as current level
Input  : levelSize as the number of entries in current 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     edgeStart ← vertices[v];
11     edgeEnd ← (v == N-1) ? M : vertices[v+1];
12     for eid ← edgeStart to edgeEnd do
13         neighbor ← edges[eid];
14         weight ← weights[eid];
15         if eid is deviation edge then
16             /* Fill out child path data     */
17             newPath ← nextLevel[offset];
18             newPath.level ← level+1;
19             newPath.from ← v;
20             newPath.to ← neighbor;
21             newPath.parent ← tid;
22             newPath.childOffset ← 0;
23             newPath.slack ← slack + distances[neighbor] +
                 weight - distances[v];
24             offset ← offset + 1;
25         end
26     end
27     /* Traverse along the shortest path
          forest                                */
28     v = forest[v];
29 end
30 return;
```

the range of `childOffset` of next level (line 5, 17, and 24). The path prefix permutation starts from the place left off by the parent (line 8) and continues along the shortest path (line 28). With the help of `childOffset`, threads are dispatched to distinct memory locations. For example, as shown in Figure 4, each thread is assigned to expand one path in *deviation level 0*. The output locations are separated in *deviation level 1* so no shared data synchronizations are required. Therefore, our algorithm is highly scalable since no explicit synchronizations are needed.

E. Intralevel Compression

Although the diameter (longest critical path) of an STA graph is typically smaller than graph size [1], the memory of GPU can be exhausted after a few iterations of expansion. Therefore, after the new level is full with critical path candidates, we perform compression to remove unwanted paths and improve memory efficiency. Assume the STA graph has vertex number N, edge number M, graph diameter d, and maximum fanout number f_{out}^{max}. The size of new level will be amplified by $O(df_{out}^{max})$ during each expansion. To compress the new level before it can be used for next expansion, we sort the new level based on path priorities (i.e., slack) and keep top k candidates with k as the number of requested paths. The `childOffset` field will be invalidated after sort but the explicit path can still be recovered by backtracing `parent`. For maximum iteration number equal to diameter d, the worst memory complexity is $O(N + M + dkf_{out}^{max})$. One part $O(N + M + dk)$ is fixed for CSR graph format and final path report. The other part $O(dkf_{out}^{max})$ is required for dynamic allocation of new level. We choose to use CPU to compress each level, because the array we sort is not large.

F. Critical Paths Recovery

Since our *deviation edge* representation is different from the explicit path trace, we have to perform critical path recovery to obtain full path trace. As the iteration expires at *maximum deviation level* (MDL), we collect our final path report by merging sorted arrays from all levels. For each *deviation edge* in the final report, we backtrace `parent` to obtain a list of *deviation edges*. Given the list, we complement the edges with *suffix edges* in the shortest path forest to recover the explicit critical path. For example, assume $e_{\emptyset C} \to e_{CE}$ in Figure 4 is selected in the final report. We recover the full path trace with forest edges in Figure 3b and obtain full path trace $CEHK$. Since the entire path recovery requires linear complexity to the requested path number k and graph diameter d, the process can be performed fast enough on CPU.

IV. Experimental Results

In this section, we demonstrate that our GPU-accelerated algorithm can efficiently generate accurate path reports with thousands to millions of paths. We conduct our experiments on a 64-bit Ubuntu Linux machine with 1 GeForce RTX 2080 GPU and 40 2GHz Intel Xeon Gold 6138 CPU cores. We compile our programs with 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. We use 1024 threads per block for all kernel configurations and 1 CPU core for all host operations. We consider the state-of-the-art path generation algorithm [3] as our baseline. To the best knowledge of the authors, the baseline has the best time complexity and practical efficiency. It has been implemented in the open-sourced STA tool, OpenTimer, as its core path generation algorithm [15], [8]. We evaluate our algorithm on real designs with a golden reference generated by an industrial standard timer [10]. To ensure fairness, we restrict our comparison to the path-based analysis part in OpenTimer.

A. Path Report Accuracy

Our algorithm can generate accurate timing report. In order to evaluate its accuracy, we use our algorithm and OpenTimer to generate separate timing reports of the same number of critical paths. We take the absolute slack difference between every pair of critical paths in two reports and record the maximum difference value. We repeat this process for different maximum deviation levels and plot our data in Figure 5. We perform this evaluation with up to a million critical paths on large benchmarks. The statistics of each benchmark are shown in Table II. As demonstrated in Figure 5, our algorithm reports almost identical critical paths as OpenTimer by expanding to 10 deviation levels on all benchmarks. This holds true even for 1M critical paths on million-gate designs leon2 (1.6M gates), leon3mp (1.2M gates), and netcard (1.5M gates). As we increase the deviation level, the accuracy keeps improving. For example, from level 2 to level 3, the maximum absolute error in 1M critical paths is reduced by over 1000 ps in most designs. Next, we compare the detailed path trace between reports. Our algorithm can report 1M critical paths exactly the same as OpenTimer on these benchmarks after 15 deviation levels of expansion. Moveover, we observe using 10 deviation levels can match 99.9% of the golden reference.

B. Runtime Performance

Our algorithm can significantly accelerate critical path generation. We compare runtime (ms) of our algorithm (1 GPU) with runtime (ms) of the PBA in OpenTimer (1 CPU core) by reporting 100K critical paths on the 10 largest benchmarks. We run our algorithm at *maximum deviation levels* (MDL) 10, 15, and 20, where our reports produced exact match with OpenTimer (see Section IV-A). Details of our experimental results are shown in Table II. We can observe that our algorithm achieves significant speed-up on million-gate designs over OpenTimer. With MDL equal to 15, we accelerate the baseline 543× on leon2 (1.6M gates), 172× on leon3mp (1.2M gates), and 304× on netcard (1.5M gates). Our algorithm also achieves 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 demonstrate our performance advantage over the baseline, Figure 6 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, and 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.

V. Conclusion

In this paper, 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. Experiments show that our algorithm achieves up to 543× speed-up on an 1.6M-gate design over the state-of-the-art PBA algorithm. At the extreme, our algorithm is 25-45×

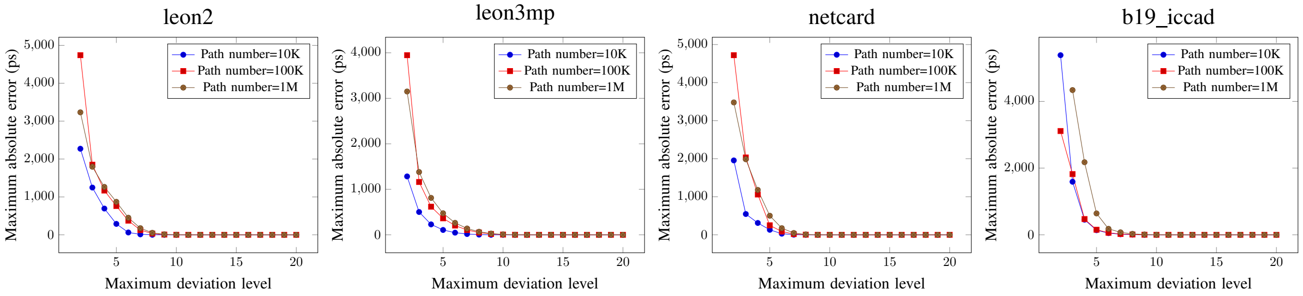


Fig. 5: Maximum absolute error between timing reports from our algorithm and OpenTimer

TABLE II: Runtime performance (ms) comparison between OpenTimer and our GPU-accelerated algorithm (1 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×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	7091.79ms	172×	8182.84	149×
netcard	3999174	1496719	7404006	752188	2050.60	367×	2475.90ms	304×	2484.08	303×
vga_lcd	397809	139529	756631	53204	682.94	77.9×	683.04ms	77.9×	706.16	75.3×
vga_lcd_iccad	679258	259067	1243041	66582	720.40	92.4×	754.35ms	88.3×	766.29	86.9×
b19_iccad	782914	255278	1576198	402645	2144.67	188×	2948.94ms	137×	3483.05	116×
des_perf_ispd	371587	138878	697145	24120	763.79	31.6×	766.31ms	31.5×	780.56	30.9×
edit_dist_ispd	416609	147650	799167	614043	1818.49	338×	2475.12ms	248×	2900.14	212×
mgc_edit_dist	450354	161692	852615	694014	1463.61	474×	1485.65ms	467×	1493.90	465×
mgc_matric_mult	492568	171282	948154	214980	994.67	216×	1075.90ms	200×	1113.26	193×

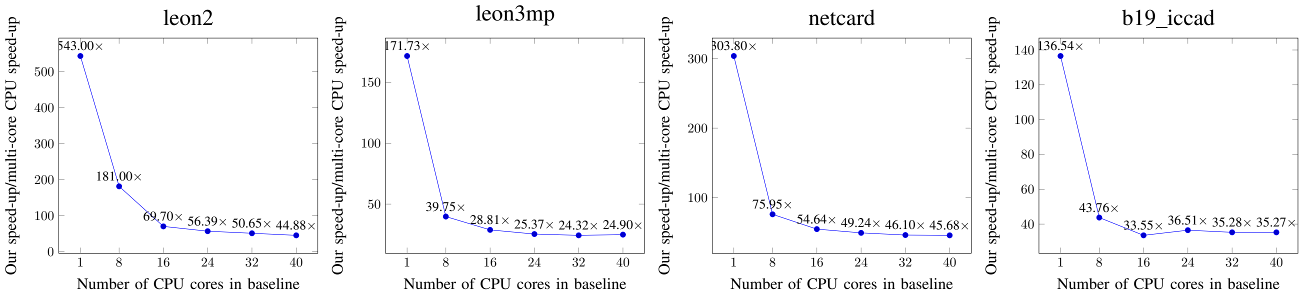


Fig. 6: Speed-up values of our algorithm over the baseline at different numbers of CPU cores.

faster than the baseline of 40 cores on million-gate designs. 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*. 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. 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.

[4] 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.

[5] 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.

[6] 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.

[7] F. Peng, C. Yan, C. Feng, J. Zheng, S. Wang, D. Zhou, and X. Zeng, "A General Graph Based Pessimism Reduction Framework for Design Optimization of Timing Closure," in *ACM/IEEE DAC*, 2018, pp. 1–6.

[8] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2020.

[9] Z. Guo, T. W. Huang, and Y. Lin, "GPU-Accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020, pp. 1–9.

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

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

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

[13] P. Harish *et al.*, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *HiPC*. Springer, 2007, pp. 197–208.

[14] P. J. Martín *et al.*, "CUDA Solutions for the SSSP Problem," in *ICCS*. Springer, 2009, pp. 904–913.

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