Fast STA Graph Partitioning Framework for Multi-GPU Acceleration

Guannan Guo*, Tsung-Wei Huang†, 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 and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

Abstract—**Path-based Analysis (PBA) is a key process in Static Timing Analysis (STA) to reduce excessive slack pessimism. However, PBA can easily become the major performance bottleneck due to its long execution time. To overcome this bottleneck, recent STA researches have proposed to accelerate PBA algorithms with manycore CPU and GPU parallelisms. However, GPU memory is rather limited when we compute PBA on large industrial designs with millions of gates. In this work, we introduce a new endpoint-oriented partitioning framework that can separate STA graphs and dispatch the PBA workload onto multiple GPUs. Our framework can quickly identify logic overlaps among endpoints and group endpoints based on the size of shared logic. We then recover graph partitions from the grouped endpoints and offload independent PBA workloads to multiple GPUs. Experiments show that our framework can largely accelerate the PBA process on designs with over 10M gates.**

I. Introduction

As the size of circuit graph grows rapidly over recent years, Static Timing Analysis (STA) has become a major performance bottleneck in design automation tools, especially when *path-base analysis* (PBA) is enabled to reduce unwanted pessimism [1]. Although PBA can improve quality of design closure, it is daunting due to its extremely long runtime. PBA needs to select a set of critical paths from an exponential number of path candidates and reanalyze them with path-specific update. As a result, the timing community never stops looking for new acceleration paradigms to alleviate the runtime challenge of PBA.

Recently, we have seen many multithreading STA paradigms [2], [3], [4], [5], [6], [7], [8], [9], [10] on multiple CPU cores. These works either exploit parallelism after circuit graph is levelized or leverage task-based models to explore top-down parallel decomposition strategies. With either method, these CPU multithreading paradigms hardly scale beyond 16 threads. Then works [11], [12] propose to advance the acceleration architectures from CPU to GPU. The proposed GPU acceleration framework can help the STA engine to achieve massive speed-up on a single GPU. However, in practice, such speed-up may not be accessible due to the limited GPU memory, since industrial designs are orders of magnitude larger. It is very difficult to offload the entire circuit graph with millions of gates, not to mention millions of paths for the PBA workload. A natural solution is to partition the circuit graph to reduce space complexity, but partitioning can also take extremely long time if we operate directly on the circuit graph. Direct partitioning will outweigh the advantage of GPU acceleration.

Therefore, in this work, we propose a fast endpoint-oriented partitioning framework for the PBA workload. In this framework, we can quickly process the STA graph and build an endpoint graph based on the size of shared logic. Then we separate the endpoint graph into groups and recover the partitions. We construct these partitions so that we can run independent STA (especially PBA) workload. Furthermore, with our GPU device manager, we can accelerate the PBA process on multiple GPUs. We highlight our main contributions as follows:

- **Fast and generic STA graph partitioning framework**. We propose an endpoint-oriented partitioning framework that can separate the STA graph based on shared endpoint logic. It takes linear time to construct an endpoint graph that reflects the size of shared logic between endpoints. Then we use family of METIS [13] graph partitioning method to separate the endpoints into groups. We recover the full partitions by a linear recovery process. Since the number of endpoints is orders of magnitude smaller than the circuit graph size, we can obtain the partitioning results much faster.
- **Independent and balanced PBA workload decomposition**. We decompose the PBA workload over the entire circuit graph into independent analysis workloads on different partitions. With our endpoint-oriented partitioning method, endpoints with larger shared logic are grouped together in the same partition, which allows us to propagate shared timing information in a single partition. If there exists shared logic across different partitions, we make minimal duplications to ensure we can conduct independent analysis.
- **Efficient GPU device management**. In case of more partitions than the number of GPU devices, we use a task-parallel approach to manage the workload balance between multiple GPU devices. We associate one CPU core to monitor the execution status of one GPU device. We leverage the work-stealing scheduler to move the imbalance analysis workload between GPU devices. In this way, we avoid idle GPU devices until all the PBA workloads are complete.

We test our framework on a set of real circuit designs provided by an industrial timer [14]. We verify the accuracy of our framework with this golden reference. To test the scalability, we also generate a set of much larger circuit designs by combining these real designs in the contest manner. We use the state-of-the-art critical path generation algorithm [10] as our CPU baseline. In our experiments, we demonstrate we can partition the STA graph into balanced pieces for independent analysis. Our framework only introduces minimal cost to obtain these partitions. In terms of performance, we achieve 8.58–24.03× speed-up over the CPU baseline on designs with over 10M gates. To the extreme, our framework can partition and analyze a 50M-gate design within an hour, while it will take the CPU baseline more than 24 hours. We believe our framework can make GPU-accelerated STA more accessible on industrial designs.

II. Proposed Partitioning Framework for Multi-GPU Acceleration

Figure 1 shows the overview of our STA graph partitioning framework for multi-GPU acceleration. First we preprocess the circuit graph by ranking the circuit pins based on their distances from the source pins. Then we construct an endpoint graph that reflects the size of common logic between endpoints. Based on this endpoint graph, we identify independent components. Each independent component is represented by a set of endpoints. If the sizes of these disjoint sets are balanced, we can simply pack the sets together. Otherwise, we have to separate the endpoint graph with METIS [13] graph partitioning method. With each endpoint graph partition, we can recover the original STA graph partition by the linear Breadth First Search (BFS) algorithm. We leverage a multithreading library, Taskflow [4], [8], to offload and monitor the PBA workloads of all partitions on multiple GPU devices.

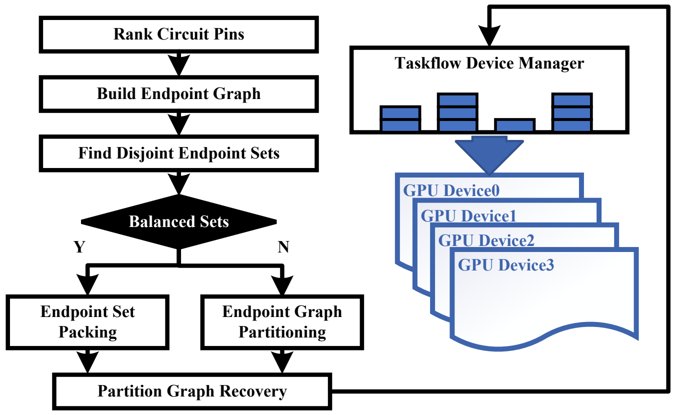


Fig. 1: Overview of our STA graph partitioning framework for multi-GPU acceleration.

A. GPU-accelerated PBA

PBA is a pivotal step in STA to remove excessive slack pessimism [1] . PBA realizes the pessimism removal by path specific update. So the critical path generation is a fundamental routine in PBA. However, the path generation process is intrinsically sequential with conventional CPU-based algorithms. Related CPU multithreading solution hardly scales beyond a few threads. Recently, Guo [12] proposes a new throughput-driven approach that harnesses the GPU computation power to explore critical paths with massive parallelism. Different from CPU-based algorithm that processes one endpoint at a time, this approach treats all endpoints as roots and unifies all the path suffix information in a forest. Then critical paths can be collected by exploring thousands of path prefixes simultaneously. Explored candidates are refined via GPU sorting algorithm and prepared for the next round of exploration. Details can be referred to the full paper [12]. Though this work demonstrates its acceleration on 1.6M gate designs, it may exhaust GPU's memory on designs with over 10M gates. So we are motivated to make GPU acceleration possible on larger designs with our partitioning framework.

B. STA Graph Model

Like most STA engines, we model the circuit graph as a directed acyclic graph (DAG). Each vertex represents a circuit pin and each edge represents a pin-to-pin connection. An endpoint represents ending pin of the datapath, usually a flip-flop input or primary output. We choose the Compressed Sparse Row (CSR) as our graph representation, because CSR is commonly used as a condensed graph format in GPU applications [15]. The CSR format contains three linear arrays which hold information for vertex offsets, edge destinations, and edge weights respectively. We also use CSR in our partitioning framework to avoid cost of conversion. We save fan-in STA graph G^{-} and fan-out STA graph G^{+} which allow us to search in both directions.

C. Circuit Pin Ranking

We first run a circuit pin ranking step to set up the weights of endpoint graph. Each weighted edge of the endpoint graph indicates the size of shared logic. To estimate the size of shared logic, we use the minimum distance of each vertex to the source pins, denoted as `ranks`. Algorithm 1 outlines this circuit pin ranking step. We initialize queue and visited list with all the source pins. Ranks of source pins are assigned with zeros. Then we explore all the neighbor vertices in breadth-first manner (line 13–17). In the meantime, we propagate the rank values plus one to the neighbors (line 18). The overall complexity of this algorithm is linear to the STA graph size, because we only conduct linear scan to the STA graph for once.

D. Endpoint Graph Construction

After we set up the ranks of all vertices, we can construct the endpoint graph in linear time. In the meanwhile, we also build disjoint sets of endpoints to identify independent components of the STA graph. As shown in Algorithm 2, we initialize the empty endpoint graph `edptGraph` with all endpoints as its vertices (line 1). Then we iterate each endpoint in the list (line 2). For each endpoint, we scan its fan-in logic cone (line 24–32). During this scan, the endpoint will propagate its index as label (line 23). Whenever we encounter a vertex that has been labeled by the other endpoint (line 9 and 10), we

Algorithm 1: Pin Ranking Algorithm

```
   Input  : Fan-out STA graph $G^+$ in CSR with $N$ vertices
            and $M$ edges, vertices[$N$], edges[$M$]
   Input  : List of source pins with $N_{source}$ as number of
            source pins, sources[$N_{source}$]
   Result: Rank table, ranks[$N$]
1  /* Initialize queue and visited list with
      all sources                                   */
2  queue ← sources;
3  visited ← sources;
4  for source in sources do
5      ranks[source] ← 0;
6  end
7  while queue.empty() is false do
8      v ← queue.front();
9      queue.pop();
10     vrank ← ranks[v];
11     edgeFront ← vertices[v];
12     edgeBack ← (v == N-1) ? M : vertices[v+1];
13     for edgeid ← edgeFront to edgeBack do
14         n ← edges[edgeid];
15         if n not in visited then
16             visited.insert(n);
17             queue.push(n);
18             ranks[n] ← vrank + 1;
19         end
20     end
21 end
22 return;
```

add edges in both directions connecting these two endpoints (line 12 and 13) in our endpoint graph. The weight of this endpoint edge is the rank of the intersection vertex, which represents the logic depth starting from source pins. Besides endpoint graph construction, we also compute the disjoint sets by merging the labels (line 14–19). After we encounter the first intersection, we skip to the next endpoint (line 21). There may be more intersections if we continue to scan, but we prioritize intersection deep in the logic. We can reduce the complexity of our endpoint graph, which saves the effort for partitioning in the later step. We also save the overall runtime in this step by early stopping.

Figure 2 illustrates an example of the endpoint graph construction. We begin with a simple STA graph in Figure 2a, where vertices $\{1, 2, 3, 4\}$ are the sources pins or startpoints, and vertices $\{13, 14, 15, 16\}$ are the endpoints. We first scan fan-in logic cone of endpoint 13. It will propagate 13 as label to vertices $\{1, 2, 5, 9, 13\}$ but it finds no intersection. Then we scan endpoint 14 and we find intersection with endpoint 13 at vertex 5. So we will add bidirectional endpoint edge in the endpoint graph with weight `rank(5)`. Then endpoint 15 intersects endpoint 14 at vertices $\{7, 10\}$ and endpoint 13 at vertex 5. However, since we prioritize the first intersection with higher weight, we only add the bidirectional edge to endpoint 14 with weight `rank(10)`. We repeat this process for endpoints 15 and 16. After this process completes, we will obtain the endpoint graph in Figure 2b. Since the endpoint graph is consisted of only STA graph endpoints as vertices,

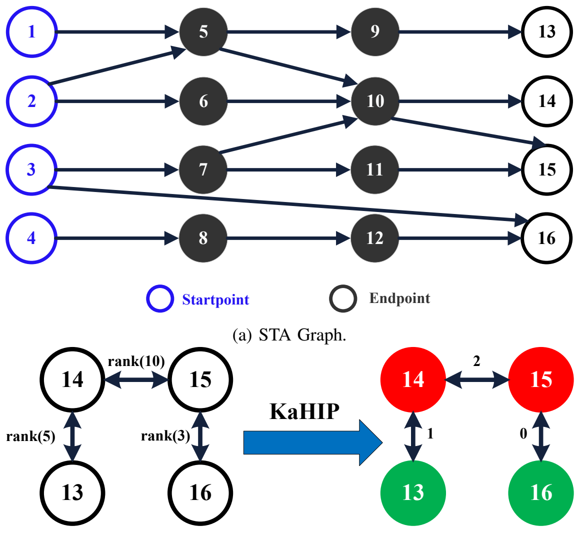
Algorithm 2: Endpoint Graph Construction Algorithm

```
   Input  : Fan-in STA graph $G^-$ in CSR with $N$ vertices and
            $M$ edges, vertices[$N$], edges[$M$]
   Input  : List of endpoints with $N_{edpt}$ as number of
            endpoints, endpoints[$N_{edpt}$]
   Input  : Rank table, ranks[$N$]
   Input  : Label table initialized with zeros, labels[$N$]
   Result: Endpoint map, edptMap[$N_{edpt}$]
   Result: Endpoint graph, edptGraph
1  edptGraph.add_vertices(endpoints);
2  for edpt in endpoints do
3      /* Initialize queue and visited list
          with current endpoint                     */
4      queue ← edpt;
5      visited ← edpt;
6      while queue.empty() is false do
7          v ← queue.front();
8          queue.pop();
9          label ← labels[v];
10         if label > 0 then
11             neighborEdpt ← label;
12             edptGraph.add_edge(edpt, neighborEdpt,
                 ranks[v]);
13             edptGraph.add_edge(neighborEdpt, edpt,
                 ranks[v]);
14             if neighborEdpt in edptMap then
15                 edptMap[edpt] ← edptMap[label];
16             end
17             else
18                 edptMap[edpt] ← label;
19             end
20             /* Jump to the next endpoint        */
21             continue;
22         end
23         labels[v] ← edpt;
24         edgeFront ← vertices[v];
25         edgeBack ← (v == N-1) ? M : vertices[v+1];
26         for edgeid ← edgeFront to edgeBack do
27             n ← edges[edgeid];
28             if n not in visited then
29                 visited.insert(n);
30                 queue.push(n);
31             end
32         end
33     end
34 end
35 return;
```

partitioning over this endpoint graph takes much less time. In this example, all endpoints have at least one intersection with some other endpoint. So there will only be one disjoint set containing all endpoints, which suggests that there is no fully independent component. In terms of complexity, we can complete the endpoint graph construction in linear time as well since we do not scan any part of the STA graph for more than once.

E. Endpoint Graph Partitioning and Recovering

In this step, we partition the endpoint graph in the previous step and recover the STA graph partition. Partitioning weighted graphs like Figure 2b is a well studied topic in graph partitioning. We choose Karlsruhe High Quality Partitioning



(a) STA Graph.

(b) Endpoint Graph.

Fig. 2: Example of endpoint graph construction.

(KaHIP) [16] because it supports CSR format as graph input and guarantees balanced partitions under some imbalance parameter. Assume we want to partition the endpoint graph in Figure 2b into two. Since endpoints 14 and 15 are connected with the highest weight, we can easily tell that the two partitions are $\{14, 15\}$ and $\{13, 16\}$. For each group of endpoints, we treat these endpoints as roots and perform BFS to obtain the entire fan-in cone. For the vertices that belong to multiple fan-in cones, we replicate these vertices to ensure the PBA workload can execute independently. Figure 3 shows the partitioning and recovering results. Since vertices $\{1, 2, 3, 5\}$ are the shared logic for both groups of endpoints, we replicate these vertices in the other partition. These replicated vertices or pins are the cost of our framework, so we construct our endpoint graph in a way to minimize this cost. After replication, we can run PBA on these partitions as independent tasks on the GPU. This strategy eliminates the connection between partitions. Otherwise we have to manage millions of prefix and suffix combinations and path fragments between partitions.

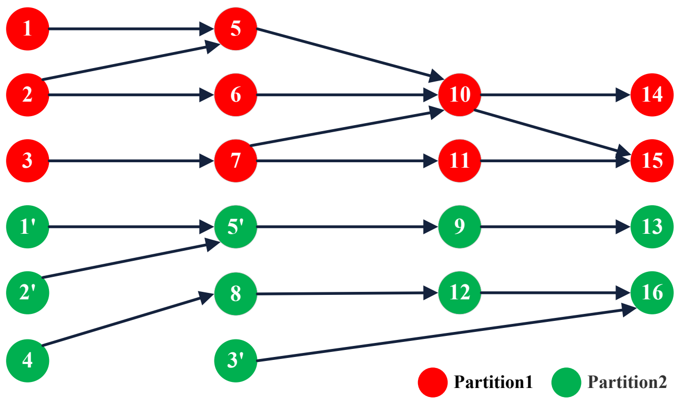


Fig. 3: Corresponding graph partitions based on the endpoint graph.

F. *Taskflow Device Management*

We leverage the work-stealing scheduler in Taskflow [4], [8] to manage multiple GPU devices. We set the number of CPU worker threads the same as the number of GPU devices. Each worker thread monitors execution of PBA on each partition. Listing 1 shows the code skeleton to use Taskflow as our GPU device manager. Assume we have processed the STA graph (line 2) and partitioned groups of endpoints (line 4). First we wrap each group of endpoints into a task (line 10–12). Then in each task, we recover the full CSR graph format from the endpoint group (line 14). We launch the PBA workload of the partition with corresponding worker index (line 15 and 16). Inside each PBA execution, we use `cudaSetDevice` on this index to select the correct GPU device. In the end, we wait for `executor` to complete all the tasks (line 20). For simplicity, we omit many details and the results handling in Listing 1. We use reduction to collect the final report from all partitions. In this way, we allow Taskflow to assist us in the GPU workload balancing.

```
1  //STA graph
2  STAGraph graph;
3  //endpoint graph partitioning results
4  std::vector<std::vector<edpt>> edptGroups;
5  //executor contains the same number of
6  //CPU workers as GPU devices
7  tf::Executor executor;
8  tf::Taskflow taskflow;
9
10 tf::Task task = taskflow.for_each(
11   edptGroups.begin(), edptGroups.end(),
12   [&](auto& group){
13     GraphCSR partCSR;
14     graph.recover_partition(group, partCSR);
15     size_t deviceID = executor.this_worker_id();
16     pathAnalysisGPU(deviceID, partCSR);
17   }
18 );
19
20 executor.run(taskflow).wait();
```

Listing 1: Taskflow device manager

III. Experimental Results

In this section, we demonstrate that our STA graph partitioning framework can quickly decompose the circuit graph and then accelerate the PBA process on multiple GPUs. We conduct our experiments on a 64-bit Ubuntu Linux machine with four GeForce RTX 2080 GPUs and four 2GHz Intel Xeon Gold 6138 CPU cores. We compile our device code with CUDA NVCC 11.0 and host code with GNU GCC 8.3.0. We use optimization flag `-O3` and C++17 standard `-std=c++17`. Our baseline algorithm is utilized in the open-sourced STA tool, OpenTimer, as its core PBA routine [10]. We run the CPU baseline on a 3.2GHz Intel i5-4670 CPU core. For experimental benchmarks, we use real designs with a golden reference generated by an industrial standard timer [14]. To further demonstrate our performance advantage on much larger designs, we follow the contest strategy to connect multiple million-gate designs in a mesh topology and generate a set of designs that have more than 10 million gates.

TABLE I: Runtime performance (second) comparison between the CPU baseline and our multi-GPU STA (4 GPUs)

Benchmark	#Pins	#Gates	CPU Baseline Runtime (second)	#Partitions=4			#Partitions=8		
				Replication Factor	Runtime (second)	Speed-up	Replication Factor	Runtime (second)	Speed-up
leon2_iccad	4.33M	1.62M	1.31M	1.21	155K	8.45×	1.51	155K	8.45×
netcard_iccad	4.00M	1.50M	958K	1.36	124K	7.72×	1.41	127K	7.54×
design1	11.7M	4.36M	2.32M	1.08	318K	7.30×	1.30	397K	5.84×
design2	23.4M	8.72M	4.75M	1.02	371K	12.80×	1.07	658K	7.22×
design3	27.7M	11.5M	5.94M	1.05	578K	10.28×	1.11	1.10M	5.40×
design4	35.1M	13.1M	6.46M	1.02	753K	8.58×	1.08	850K	7.60×
design5	46.8M	17.4 M	9.56M	1.01	798K	11.98×	1.02	1.05M	9.10×
design6	93.6M	34.8M	26.7M	1.01	977K	30.40×	1.02	1.46M	18.29×
design7	117M	43.6M	31.0M	1.01	1.29M	24.03×	1.03	1.66M	18.67×

A. *Runtime Performance*

Our partitioning framework can quickly separate the STA graph and completes the PBA process on multiple GPUs. We compare the performance of the CPU baseline against our framework by reporting 100K critical paths. Before comparing the performance, we first verify our framework does not induce any loss of accuracy compared to the baseline. We can achieve this because we duplicate circuit pins in the shared logic. So we also record the *replication factor* to indicate the marginal cost introduced by our framework. We define *replication factor* as the average number of times that a circuit pin shows up in all partitions combined. For example, *replication factor*=1 implies that our partitioning framework is not duplicating any pin and the STA graph is perfectly separated. We experiment with four GPUs because running with single GPU is slower and wastes parallelism from independent partitions. Table I shows an overview of our experiments. Our framework maintains an obvious performance advantage for designs ranging from 1.5M gates to 43.6M gates. We can observe that, as the size of the design increases, the margin of our advantage grows larger. We start with 7.72× speed-up on `netcard_iccad` with 1.5M gates. We then achieve 12× speed-up for designs around 10M gates. By the last design with 43.6M gates, we reach 24.03× speed-up with 4 partitions. To the extreme, we have also tested designs above 50M gates. The CPU baseline could not finish in 24 hours so the design is not provided in the table. However, our framework can complete in less than an hour.

Our framework does not restrict equal partition number with the GPU device number. It is common that we need smaller partitions to accommodate the PBA workload on GPU. Therefore, we also provide statistics for experiments with 8 partitions. Since partitioning takes the majority of the runtime, graph partitioning will cost longer runtime with more partitions. Depending on different STA graph topology, this cost varies. For example, there is nearly no runtime difference in `leon2_iccad` and `netcard_iccad`. In design3, the runtime almost doubles. However, our framework still outperforms the CPU baseline with 5.40–18.67× speed-up. Besides the promising runtime performance, we also want to highlight our low space cost. In all our experiment benchmarks, a circuit pin is not replicated more than once for 4–8 partitions. We will discuss more about the quality of the partitions in the subsequent section.

B. *Partition Quality and Cost*

Our partitioning strategy provides balanced partitions with low marginal cost. To demonstrate this, we first measure the GPU runtime of four partitions running on respective GPU devices. Figure 4 shows the runtime distribution of four GPU devices analyzing four partitions. We can observe that the GPU runtime closely matches with each other between different devices. The maximum runtime differences are 15.7% for design2, 13.2% for design3, 15.1% for design4, and 19.2% for design5. The runtime variations between devices are low compared to the averaged runtime. This runtime distribution reflects that partitions generated by our framework are balanced.

To understand the marginal cost of our partitioning framework, we plot the *replication factor* as a function of partition number between 2–10. We choose four largest and densest real designs from the golden reference[14]. These designs are very difficult to separate or partition. For example, in designs `leon2_iccad` and `leon3mp_iccad`, about 90% of the endpoints share common logic in the circuit graph. As shown in Figure 5, our partitioning framework maintains *replication factor* less than 2.0 on these designs between 2–10 partitions. Looking at the trend of *replication factor* in `leon2_iccad`, `leon3mp_iccad`, and `netcard_iccad`, we observe higher cost if we are requesting more partitions. This is expected since many endpoints are tightly grouped in these designs. An exception to this trend is `b19_iccad`, where endpoints are sparsely connected. In this case, our partitioning framework can easily capture this feature and make almost perfect partitions. In general, our framework ensures low partitioning cost on dense designs and nearly zero cost on sparse designs.

C. *Device Management and Workload Balancing*

Our Taskflow device manager balances the device workload when the number of partitions exceeds the number of devices. As shown in Figure 6, we compare the runtime distribution of GPU devices with 16 partitions against four partitions on

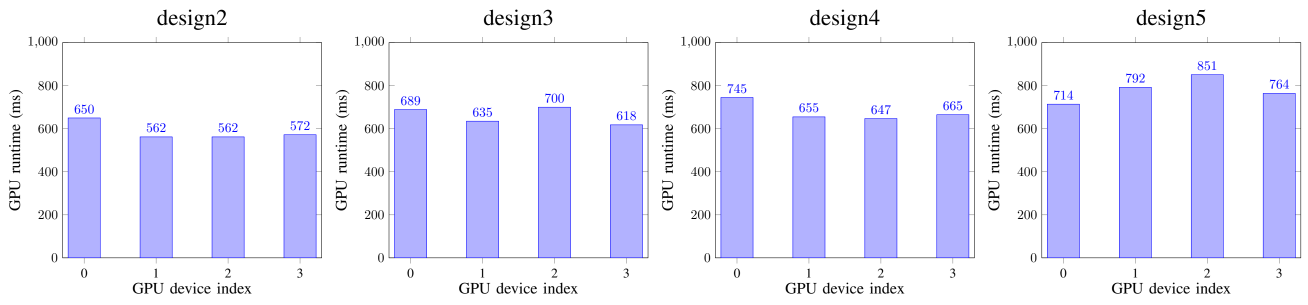


Fig. 4: Runtime distribution of four GPU devices executing PBA workload over four partitions.

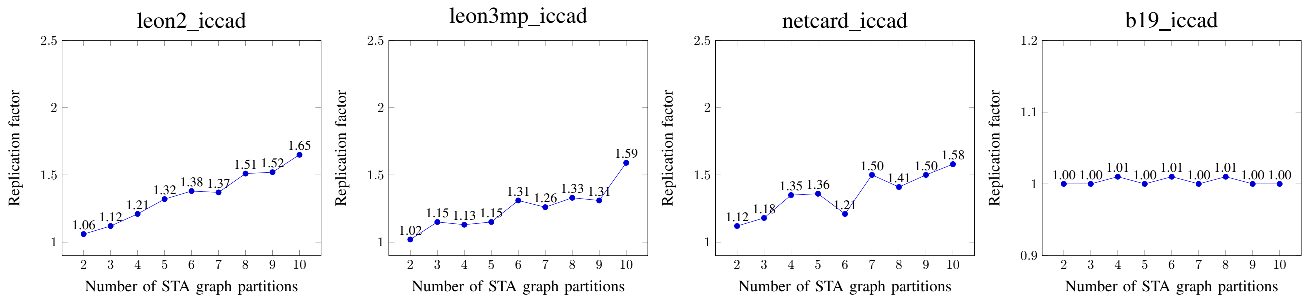


Fig. 5: Replication factor with respect to the number of STA graph partitions.

a 50M-gate design. With 16 partitions on four GPU devices, we maintain balanced execution time between devices. We can also observe overall 29% runtime increases compared to four partitions since we are invoking four times more CUDA API calls per device with 16 partitions. To summarize, our Taskflow device manager effectively balances the workloads between GPU devices.

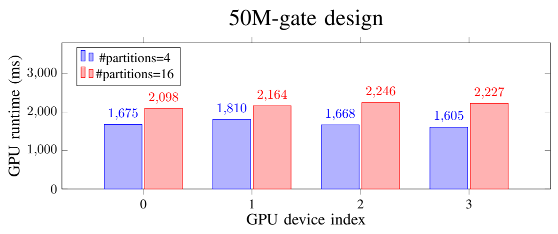


Fig. 6: Runtime distribution comparison between #partitions=4,16

IV. Conclusion

In this paper, we have introduced a fast STA graph partitioning framework that can accelerate the PBA workload on multiple GPUs. We obtain balanced graph partitions by separating an weighted endpoint graph. We ensure the PBA workloads can execute on these partitions independently. We balance the workload between GPU devices by a smart device manager. Experiments show that our framework can accelerate PBA by 12–24× for designs above 10M gates.

Acknowledgment

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523, OAC-2209957, and TI-2229304.

References

[1] J. Bhasker *et al.*, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.
[2] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, “Fast path-based timing analysis for cppr,” in *IEEE/ACM ICCAD*, 2014, pp. 596–599.
[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] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,” in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 6, 2022, pp. 1303 – 1320.
[6] C.-H. Chiu and T.-W. Huang, “Composing Pipeline Parallelism Using Control Taskflow Graph,” in *ACM HPDC*, 2022, pp. 283—284.
[7] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, “GPU-accelerated Critical Path Generation with Path Constraints,” in *IEEE/ACM ICCAD*, 2021.
[8] C.-H. Chiu and T.-W. Huang, “Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms,” in *ACM/IEEE Design Automation Conference (DAC)*, 2022.
[9] 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.
[10] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, “OpenTimer v2: A New Parallel Incremental Timing Analysis Engine,” *IEEE TCAD*, 2020.
[11] Z. Guo, T. W. Huang, and Y. Lin, “GPU-Accelerated Static Timing Analysis,” in *IEEE/ACM ICCAD*, 2020, pp. 1–9.
[12] G. Guo, T.-W. Huang, Y. Lin, and W. Martin.D.F, “GPU-accelerated Path-based Timing Analysis,” in *IEEE/ACM DAC*, 2021.
[13] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[14] T. Huang and M. Wong, “OpenTimer: A high-performance timing analysis tool,” in *IEEE/ACM ICCAD*, 2015, pp. 895–902.
[15] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *IEEE/ACM SC*, 2009.
[16] P. Sanders and C. Schulz, “Think Locally, Act Globally: Highly Balanced Graph Partitioning,” in *SEA*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.