Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System

Tsung-Wei Huang
The University of Wisconsin at Madison
Madison, Wisconsin, USA
tsung-wei.huang@wisc.edu

Boyang Zhang
The University of Wisconsin at Madison
Madison, Wisconsin, USA
bzhang523@wisc.edu

Dian-Lun Lin
The University of Wisconsin at Madison
Madison, Wisconsin, USA
dlin57@wisc.edu

Cheng-Hsiang Chiu
The University of Wisconsin at Madison
Madison, Wisconsin, USA
chenghsiang.chiu@wisc.edu

ABSTRACT

Static timing analysis (STA) is an integral part in the overall design flow because it verifies the expected timing behaviors of a circuit. However, as the circuit complexity continues to enlarge, there is an increasing need for enhancing the performance of existing STA algorithms using emerging heterogeneous parallelism that comprises manycore central processing units (CPUs) and graphics processing units (GPUs). In this paper, we introduce several state-of-the-art STA techniques, including task-based parallelism, task graph partition, and GPU kernel algorithms, all of which have brought significant performance benefits to STA applications. Motivated by these successful results, we will introduce a task-parallel programming system to generalize our solutions to benefit broader scientific computing applications.

CCS CONCEPTS

• **Hardware → Static timing analysis**.

KEYWORDS

Static timing analysis; high-performance computing; task parallelism; heterogeneous parallelism

ACM Reference Format:
Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *Proceedings of the 2024 International Symposium on Physical Design (ISPD '24), March 12–15, 2024, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626184.3635278

1 INTRODUCTION

As the circuit size and complexity continue to increase, static timing analysis (STA) has quickly become a bottleneck in the overall design flow due to its intensive yet time-consuming interaction with other design tools [8]. For instance, a timing-driven placement optimization algorithm can call a timer in a loop thousands or millions of times to evaluate the timing result of an optimization strategy. Besides, research has reported that completing a full timing analysis for a large design of multi-million gates can take several hours [35, 44]. To overcome these runtime challenges, there is an increasing need for accelerating existing STA algorithms using emerging heterogeneous parallelism that comprises manycore central processing units (CPUs) and graphics processing units (GPUs).

Recognizing this need, we have been researching and developing various parallel STA algorithms since 2014. By harnessing the power of CPU-GPU heterogeneous computing, our algorithms have successfully accelerated many time-consuming STA tasks in both graph-based analysis (GBA) [21, 24] and path-based analysis (PBA) [17–19, 22, 23, 25, 43, 45, 46] applications. To efficiently handle large designs, we also introduced new partitioning algorithms [10, 20, 44] to offload partitioned STA tasks across distributed machines [31, 34, 40, 41, 44, 47]. More importantly, we have integrated our innovations into the open-source timing analysis software, OpenTimer [29, 36, 42], to facilitate research on efficient STA algorithms. So far, OpenTimer has been used by many researchers (ICCAD CAD Contests [51], TAU Contests [33]) and vendors (DARPA IDEA [1], Motivo [3], efabless [5]) to analyze the timing of their designs.

In this paper, we will briefly discuss our solutions for parallel and heterogeneous STA algorithms. We will go through four cornerstone strategies in the next four sections: (1) *task-parallel STA algorithms* to improve the asynchrony of timing propagation in GBA, (2) *task graph partitioning* to improve the scheduling performance of task-parallel STA algorithms, (3) *GPU-accelerated STA algorithms* to speed up time-consuming PBA applications, and (4) a *task-parallel programming system* to generalize our approach for broader applications beyond STA. For each section, we will highlight the experimental results achieved by our strategy.

2 TASK-PARALLEL STA ALGORITHMS

2.1 Motivation

Most existing timers, including commercial ones, count on *loop-based parallelism* to parallelize GBA and other timing propagation tasks. In a rough view, the circuit is *levelized* to a topological order, and levels of nodes are kept in a dynamic data structure called

ISPD '24, March 12–15, 2024, Taipei, Taiwan

ACM ISBN 979-8-4007-0417-8/24/03
https://doi.org/10.1145/3626184.3635278

level list. Since all the nodes in the same level are independent on each other and can run in parallel, one can apply language-specific "`parallel for`" to each list of nodes level by level. This level-based decomposition is advantageous in its simple *task-parallel pipeline* concept [11, 12, 15] and is by far the most implementation in existing timers, including our first version of OpenTimer v1 [42].

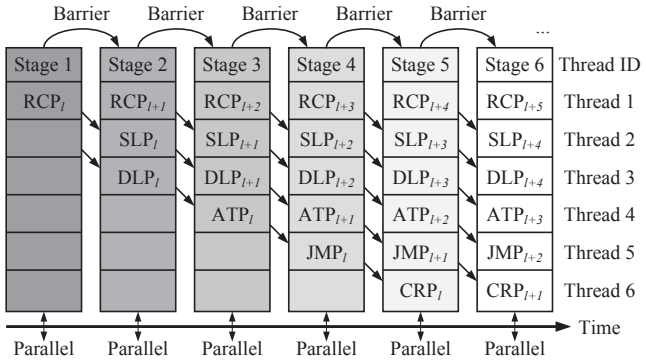

Figure 1: Loop-based parallel timing propagation. Each level applies a `parallel_for` to update timing from the fanin of each node [42].

Algorithm 1: update_timing_using_loop_parallelism()

```
1  B ← Level list of the timer;
2  if B.num_pins = 0 then
3  |   return;
4  update_level(B);
5  l_min ← B.min_nonempty_level;
6  l_max ← B.max_nonempty_level;
7  # Parallel_Region {
8  # Master_Thread_do for l = l_min to l_max + 4 do
9  |   # spawn_task propagate_rc(l);
10 |   # spawn_task propagate_slew(l − 1);
11 |   # spawn_task propagate_delay(l − 1);
12 |   # spawn_task propagate_arrivel_time(l − 2);
13 |   # spawn_task propagate_jump_point(l − 3);
14 |   # spawn_task propagate_cppr_credit(l − 4);
15 |   # synchronize_tasks;
16 };
17 # Parallel Region {
18 # Master_Thread_do for l = l_max to
     B.min_non_empty_level do
19 |   # spawn_task propagate_fanin(l);
20 |   # spawn_task propagate_required_arrival_time(l);
21 |   # synchronize_tasks;
22 };
23 remove all pins from the level list B;
```

Figure 1 illustrates this strategy on an example of forward timing propagation in OpenTimer v1 [42]. For each node, we update a sequence of dependent tasks including parasitics (RCP), slew (SLP), delay (DLP), arrival time (ATP), jump points (JMP), and common path pessimism reduction (CRP). We encapsulated the task dependency into a parallel pipeline and used one thread to run a particular type of task level by level. Details of each task can be referred to [42]. OpenTimer v1 carried out this strategy using Algorithm 1. It first calls `update_level` to update the level list for incremental timing (line 5). The timing propagation is then performed level by level in a parallel pipeline (line 8:18 for forward timing propagation and line 19:25 for backward timing propagation). By the end of each pipeline stage, a barrier is imposed to synchronize all spawned tasks (line 16 and line 23). The level list is reset after the timing propagation completes (line 26). Depending on applications, the timer may add more tasks to the pipeline for parallelism.

The loop-based pipeline strategy is simple and easy to implement using popular parallel programming libraries such as OpenMP [4] and Intel Threading Building Blocks (TBB) Flow Graph [2]. However, it suffers from many performance drawbacks. For example, the number of nodes can vary from level to level, resulting in highly unbalanced computations and thread utilization. Also, there is a synchronization barrier between successive levels in order to keep task dependencies. The overhead can be large for graph with long data paths. Furthermore, it is difficult to extend the pipeline to efficiently include other types of compute-intensive tasks such as advanced delay modeling, signal integrity, and cross-talk analysis. These tasks often expose dependency constraints across multiple layers of the timing graph that do not fit in a single level [8]. Neither can path-specific update be included to the pipeline without significant rewrite of the core data structure.

2.2 Parallel Timing Update using Task Graph

To overcome the synchronization challenge of pipeline-based timing update, we have introduced a new task graph-based timing propagation method. Instead of representing a circuit graph and storing propagation tasks in a levelized list, we represent these tasks in a *task graph* that is directly constructed from the circuit itself. Each task represents a specific timing propagation task, and each edge represents a functional dependency between two tasks. By delegating the scheduling to an efficient task graph execution runtime (e.g., work-stealing scheduler [55]), we can gain transparent parallelization and dynamic load balancing. We have implemented this strategy in OpenTimer v2 [29].

Figure 2 gives an example of a task graph-based timing update. White nodes represent forward propagation tasks (e.g., slew propagation, delay calculation, arrival time update), while black nodes represent backward propagation tasks (e.g., required arrival time update). We can clearly see the advantage of task graph-based timing update, as multiple timing propagation tasks can start immediately whenever their dependency constraints are met. There is no need to wait for the previous level to complete before moving onto the next level. Furthermore, overlap between forward and backward propagation tasks are now possible. As a result, the runtime efficiency is largely improved due to more asynchronous execution than loop-based pipeline parallelism.

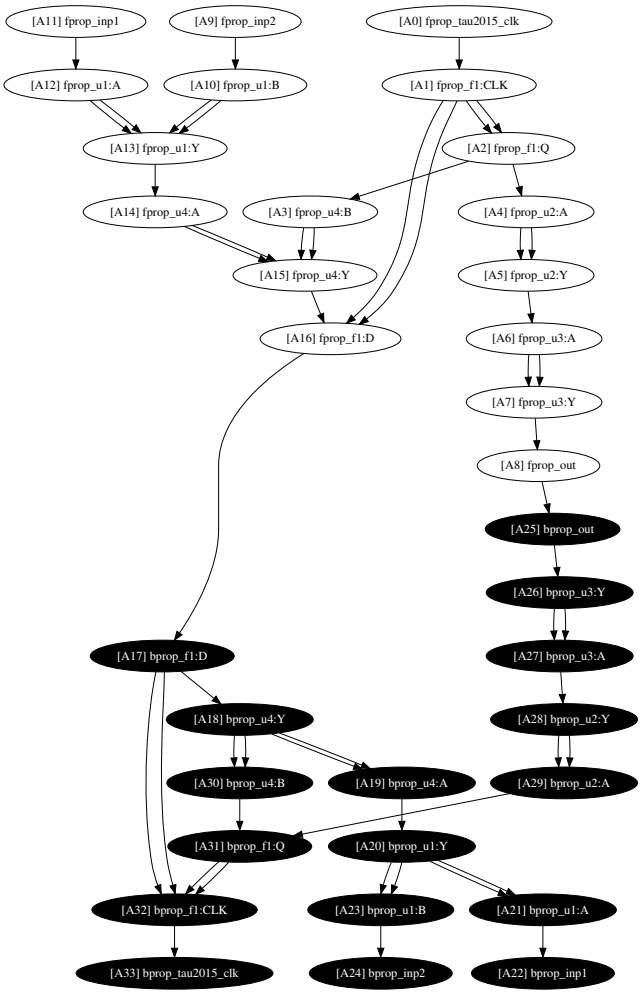

Figure 2: A task graph to carry out parallel timing update. The graph consists of forward propagation tasks (white) and backward propagation tasks (black).

2.3 Experimental Results

Figure 3 plots the runtime results at different numbers of CPU cores for two different parallel timing propagation algorithms, *loop parallelism* and *task parallelism*, which were implemented in OpenTimer v1 [42] and OpenTimer v2 [29], respectively. We observed task parallelism scales higher than loop parallelism with increasing cores. Both saturate at about 8-12 cores. The saturation is affected by many factors such as the graph structure and the size of timing propagation tasks.

Regardless of the core count used, task-parallel strategy leads to much better runtime scalability over loop-based parallelism. The maximum parallelism in the loop-based strategy is dominated by the number of independent nodes in a single level. Computing nodes between different levels incurs a synchronization cost. By contrast, task-parallel strategy lets computation flow naturally with the timing graph structure. The scheduler autonomously optimizes the parallelism with dynamically generated tasks. As a consequence,

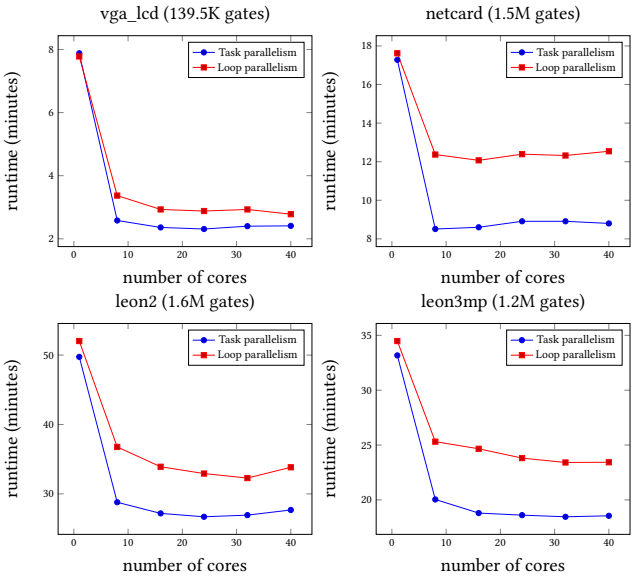

Figure 3: Comparison of runtime scalability between loop-parallel and task-parallel timing propagation algorithms over increasing number of CPU cores on four large circuits, vga_lcd, netcard, leon2, and leon3mp.

the runtime difference between the two strategies becomes remarkable as we increase the numbers of cores and tasks. For example, it took 12.54 minutes for loop-based method to finish whereas task-parallel implementation reached the goal in only 8.80 minutes (30% faster). At about 8–16 cores where both methods reach the saturation point, the gap between the two remains pronounced.

3 TASK GRAPH PARTITION

3.1 Motivation

While using task dependency graph (TDG) brings significant speed up to STA, the scheduling cost, which includes the construction and execution of a TDG, may become predominant when handling extensive STA workloads. For example, the analysis of a circuit with 1.5M gates can allocate more than 50% of the runtime to construct the corresponding TDG with 4M tasks and 5M dependencies [6]. However, the TDG execution performance speedup often saturates under only 8-16 CPU threads. This outcome indicates it is unnecessary to have a large TDG, considering the limited number of saturated CPU threads. Moreover, the majority of timing propagation tasks have very short runtime compared to the per-task scheduling cost. For example, a backward propagation task in OpenTimer [6] takes about 0.5-50 us, while scheduling such a task using OpenTimer's Taskflow scheduler [7] can task 0.2-3 us. Therefore, it is important to achieve a balance between scheduling cost and task granularity to optimize the performance of task-parallel STA algorithms.

A typical approach to minimize scheduling cost is to partition a large TDG into numerous clusters. Each cluster comprises tasks that run sequentially based on their topological order in the original TDG. Instead of scheduling individual tasks across various workers,

we now schedule a partition only once and execute it with a single worker, thereby reducing scheduling overhead. Figure 4 shows the advantage of TDG partitioning by recording the runtime of the core "update_timing" method in OpenTimer [6]. We assume the TDG maintains the same topology throughout different iterations as this is a common scenario in STA (e.g., gate sizing). Therefore, partitioning is only conducted once at the first iteration of STA. From Figure 4, we can see that despite the partitioning overhead in the first couple of iterations, partitioning largely improves the STA runtime performance due to the reduction of scheduling overhead in the later iterations. The improvement continues to accumulate as we increase the number of STA iterations.

There have been various TDG partitioning algorithms. Vivek [64] partitions tasks based on their impact on critical path length and TDG parallelism. However, their partitioning algorithm requires iterative cycle checking, which results in quadratic time complexity. To improve time complexity, GDCA [9, 63] removes iterative cycle checking by partitioning the TDG based on its topological order. However, this largely impacts the partitioned TDG parallelism. Moreover, they are all limited to single-threaded execution. As the size of the TDG increases, their partitioning time grows significantly, potentially negating the benefits of partitioning.

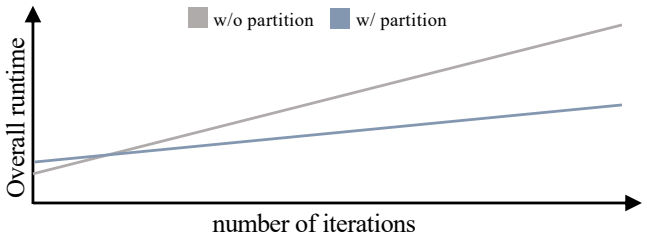

Figure 4: Overall STA runtime of the core "update_timing" method in OpenTimer [6] under different timing update iterations with and without partitioning.

3.2 CPU-parallel Partitioning Algorithm

To expedite the TDG partitioning process, we propose C-PASTA, a CPU-parallel partitioning algorithm for STA. C-PASTA not only achieves significant speedup in partitioning but also minimizes the reduction of TDG parallelism by prioritizing clustering tasks between adjacent levels. Moreover, to eliminate the time-consuming cycle-checking process, C-PASTA incorporates a simple yet efficient cycle-free clustering algorithm that restricts the parent partition to which a task can be clustered.

Algorithm 2 describes the overall partitioning process for C-PASTA. We maintain a work-stealing queue for each CPU thread. At the beginning of partitioning, all the source tasks of the TDG are pushed into the queue of the first CPU thread. During partitioning, each CPU thread first works on the tasks in its own queue until the queue is empty, then steals a task from one of the other threads' queues. When a CPU thread is working on a task, it first assigns the partition for this task using our cycle-free clustering algorithm, then checks if this task initiates a linear chain of tasks, meaning there is only one dependency between two tasks in adjacent levels. If such a linear chain exists, the CPU thread iteratively traverses the tasks on this chain and assigns their partitions. Finally, upon reaching the end of the linear chain, the CPU thread releases the dependents for the neighboring tasks of the current task by incrementing their dependent counters. Once the dependent counter of a neighboring task equals its number of dependents, that task is placed into the queue for later processing.

Algorithm 3 describes C-PASTA's cycle-free clustering algorithm. Specifically, we assign a partition ID for each task in the TDG. In cases where multiple partitions seek to cluster one task, only the partition with the largest ID is able to cluster that task. Since C-PASTA traverses the TDG in a top-down manner, the partition IDs of all tasks at one level are always larger than those at previous levels. This enforces no cyclic dependencies will be introduced during partitioning because the partition with a larger ID will always come after a partition with a smaller ID. Once the largest partition ID from the dependents of the to-be partitioned task is obtained, we increment the partition counter of this partition which counts the number of tasks within this partition. Then, we verify whether the partition counter surpasses the user-defined partition size (*P_S*), indicating the maximum allowable number of tasks for a partition. If so, we assign the to-be partitioned task a new partition ID by incrementing the current maximum partition ID (*max_partition*) by one.

Figure 5 shows an example of C-PASTA's partitioning algorithm in three iterations. In the first iteration, all the source tasks 0, 1, and 2 are pushed into the work-stealing queue of thread 1. While thread 1 is working on task 2, tasks 0 and 1 are stolen and processed by threads 2 and 3 separately. Moving to the second iteration, threads 1, 2, and 3 handle tasks 5, 3, and 4 along the linear chains. Notably, tasks on the linear chains are directly processed by threads without being enqueued. After the second iteration, the dependents of task 6 are all released and task 6 is pushed into the queue of thread 1. Thread 1 processes task 6 in the final iteration. The partitioning result is shown in Figure 5(c).

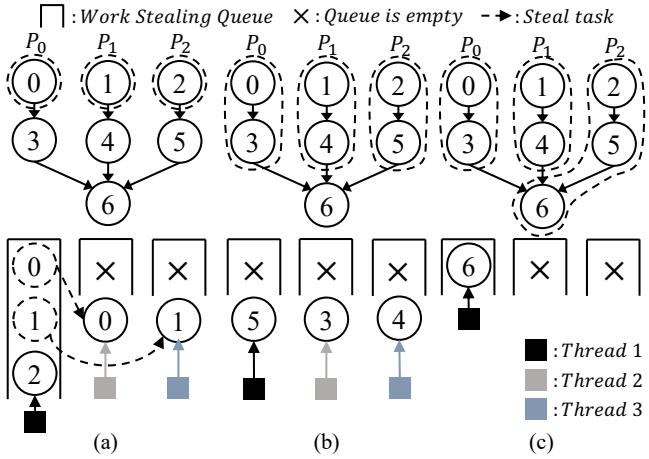

Figure 5: An example of C-PASTA's partitioning algorithm in three iterations under the partition size of 3.

Algorithm 2: Overview of C-PASTA

```
1  push_source_tasks_to_queues();
2  parallel for each thread tid {
3  while task_cnt.load() < total_num_task do
4      /* step 1. Process the tasks in the local queue */
5      while !queues[tid].empty() do
6          task = queues[tid].pop();
7          task_cnt.fetch_add(1);
8          cycle_free_clustering(task);
9          while is_linear_chain(task) do
10             task = task.neighbor;
11             cycle_free_clustering(task);
12         for n ∈ task.neighbor do
13             if n.dep_cnt.fetch_add(1) = n.num_deps then
14                 queues[tid].push(n);
15     /* step 2. Steal a task from other queues */
16     steal_from = traverse_queues_to_steal();
17     task = queues[steal_from].steal();
18     task_cnt.fetch_add(1);
19     cycle_free_clustering(task);
20     while is_linear_chain(task) do
21         task = task.neighbor;
22         cycle_free_clustering(task);
23     for n ∈ task.neighbor do
24         if n.dep_cnt.fetch_add(1) = n.num_deps then
25             queues[steal_from].push(n);
26 }
```

Algorithm 3: cycle_free_clustering(*task*)

```
1 desired_partition = get_max_partition(task.dependents);
2 if partition_cnt[desired_partition].fetch_add(1) < P_S then
3     task.partition = desired_partition;
4 else
5     new_partition = max_partition.fetch_add(1) + 1;
6     task.partition = new_partition;
7     partition_cnt[new_partition]++;
```

3.3 Experimental Results

For partitioning performance comparison, We consider GDCA as our baseline due to its efficiency. As GDCA requires users to provide a partition size, we select the optimal partition size that produces the best performance for each circuit; for C-PASTA, we simply use the TDG size for the partition size, as our algorithm will converge to a suitable value. We conduct our experiments by integrating GDCA and C-PASTA into OpenTimer [6] and run graph-based analysis (`update_timing` command) on five industrial circuits.

Table 1 compares the overall performance between GDCA and C-PASTA in terms of their runtime improvements on the generated TDGs and their partitioning runtime. The values under the T_{TDG^P} column show the runtime of partitioned TDGs and their speedup over the original TDGs. The values under the $T_{Partition}$ column show the runtime of C-PASTA and GDCA and C-PASTA's

Table 1: Overall performance comparison between GDCA and C-PASTA and their improvements on generated TDGs in the core `update_timing` method of OpenTimer [6].

circuit	#tasks	#deps	T_{TDG} (ms)	T_{TDG^P} (ms)		$T_{Partition}$ (ms)	
				GDCA	C-PASTA	GDCA	C-PASTA
des_perf	303.7K	387.3K	25.5	15.9 (1.6×)	13.5 (1.8×)	40.3	17.3 (2.3×)
vga_lcd	397.8K	498.9K	34.7	21.3 (1.6×)	19.8 (1.7×)	57.9	22.5 (2.5×)
leon3mp	3.4M	4.1M	286.0	148.9 (1.9×)	133.7 (2.1×)	567.1	179.0 (3.1×)
netcard	3.9M	4.9M	338.3	173.6 (1.9×)	155.9 (2.1×)	656.1	155.0 (4.2×)
leon2	4.3M	5.3M	372.6	192.0 (1.9×)	180.5 (2.0×)	762.0	221.9 (3.4×)

speedup over the baseline GDCA. We measure the performance in one full-timing iteration through the `update_timing` method in OpenTimer [6]. Overall, both GDCA and C-PASTA can improve the performance of `update_timing` due to reduced TDG size and scheduling cost. However, C-PASTA always outperforms GDCA. For instance, C-PASTA can improve the TDG runtime of the five circuits by 1.7–2.1×, whereas GDCA is 1.6–1.9×. We attribute this result to C-PASTA's effort to minimize the impact on the original TDG parallelism during partitioning. In terms of partitioning runtime, C-PASTA surpasses GDCA on all five circuits. The largest speedup values are observed in netcard where C-PASTA is 4.2× faster than GDCA.

Figure 6 compares the TDG runtime (after partitioning) between GDCA and C-PASTA under different partition sizes. Note that the partition size refers to the maximum number of tasks within a partition. As GDCA strictly requires each partition to have the same size, its TDG runtime shows a V-shape pattern, where the runtime first decreases because of reduced scheduling cost and then increases because of reduced parallelism. For GDCA, users are responsible for finding the right partition size that produces the best performance. However, for C-PASTA, the TDG runtime continues to decrease until saturation. This is because Algorithm 3 always maintains a minimum number of partitions during partitioning, ensuring a lower limit for the resultant TDG parallelism. This characteristic emphasizes an additional benefit of C-PASTA, eliminating the necessity for users to fine-tune the partition size. Instead, they can effortlessly employ the original TDG size as the default value. C-PASTA will autonomously converge to the optimal partition size and granularity, ensuring optimal TDG runtime performance.

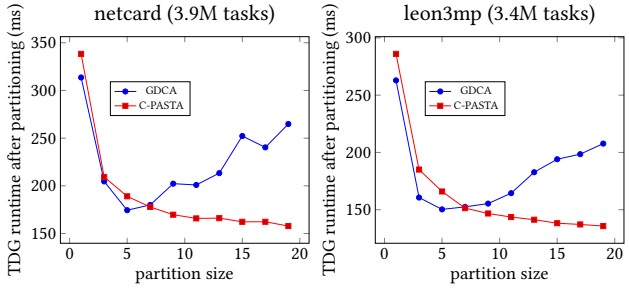

Figure 6: Comparison of TDG runtime (after partitioning) between GDCA and C-PASTA under different partition sizes.

4 GPU-ACCELERATED STA ALGORITHMS

4.1 Motivation

PBA is pivotal for achieving accurate timing results by reducing unwanted pessimism in STA [8]. However, PBA is extremely time-consuming, typically 10-1000× slower than graph-based analysis GBA [43]. 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 [26, 32, 44, 50, 53, 62]. 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 [32, 44] adopts task-based parallelism with exact accuracy, but its performance saturates at 16 cores. Similarly, Jin [50] and Peng [62] propose fast and accurate block-based algorithms that improve runtime up to 1.63×, but their algorithms are highly sequential and scales to fewer CPU cores.

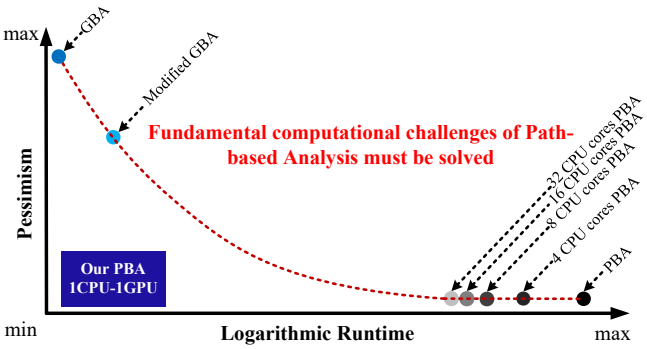

Figure 7: Computational trade-off between runtime and pessimism reduction on path-based timing analysis [18, 19].

As illustrated in Figure 7, 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.

4.2 GPU Kernels for Path Enumeration

Figure 8 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 an 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.

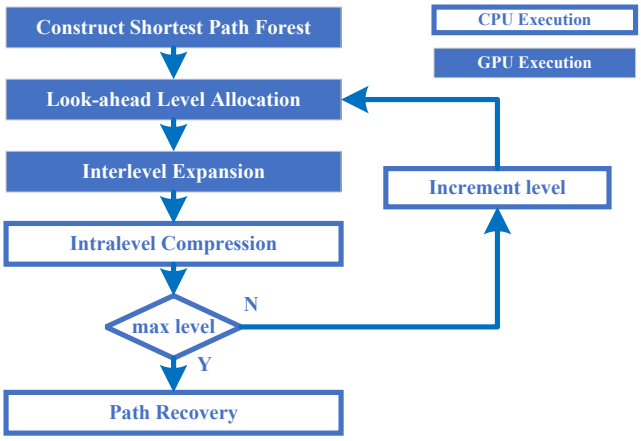

Figure 8: Overview of our GPU-accelerated PBA algorithm [19].

4.3 Experimental Results

To demonstrate our performance advantage over the baseline, Figure 9 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 45.68× and 35.27× on two large designs 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.

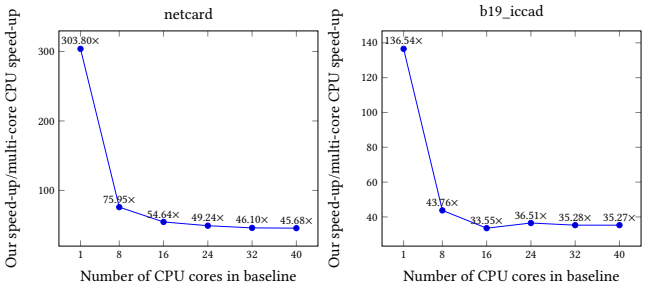

Figure 9: Speed-up values of our GPU-accelerated PBA algorithm over a CPU baseline.

5 TASK-PARALLEL PROGRAMMING SYSTEM

Given these successful results, we have further developed a new task-parallel programming systems called Taskflow [32, 37] that generalizes our solutions to benefit broader applications beyond STA. Some examples include machine learning [13, 48, 49, 54, 57–59], hardware fuzzing [61], logic simulation [16, 60], quantum computing [28], physical design [27, 30, 38, 39], power grid analysis [56], macro modeling [52], graph partitioning [10], etc. The main goal of Taskflow is to simplify the building of parallel programs using a transparent and concise graph description model. At the time of this writing, Taskflow has been used by thousands of academic and industrial projects. There are two type of task graph programming models, *static task graph programming* (STGP) and *dynamic task graph programming* (DTGP), which we explained below:

5.1 Static Task Graph Programming

STGP defines separate graph construction and graph execution. Applications need to describe the task graph first and then submit it to a Taskflow scheduler for execution. STGP is useful when the task graph structure is known at programming time and does not depend on runtime variables.

```
tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = tf.emplace(
  [] () { std::cout << "Task A\n"; },
  [] () { std::cout << "Task B\n"; },
  [] () { std::cout << "Task C\n"; },
  [] () { std::cout << "Task D\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C
executor.run(taskflow).wait();
```

Listing 1: A static task graph programming example of Taskflow [37].

Listing 1 presents an STGP example. The code is *self explanatory*. The program creates a task dependency graph of four tasks, `A`, `B`, `C`, and `D`, where task `A` runs before task `B` and task `C`, and task `D` runs after task `B` and task `C`. Each task is described as a *callable object*, which can be either a lambda, a functor, a binding expression, or an operator. Taskflow provides an abstraction over difficult concurrency controls such as threading and scheduling. Users describe an application in terms of *tasks* rather than threads. They do not need to manage threads or locks, and can focus on high-level development work. Meanwhile, Taskflow will handle all the scheduling details and achieve dynamic load balancing using work stealing [55].

5.2 Dynamic Task Graph Programming

Unlike STGP, DTGP overlaps graph construction with graph execution. Applications describe the task graph on the fly from an Taskflow executor. DTGP is useful when the task graph structure is unknown at programming time and depends on runtime variables or control-flow results.

```
tf::Executor executor;
auto [A, FuA] = tf.dependent_async([]() {
  std::cout << "Task A\n";
});
auto [B, FuB] = tf.dependent_async([]() {
  std::cout << "Task B\n";
}, A);
auto [C, FuC] = tf.dependent_async([]() {
  std::cout << "Task C\n";
}, A);
auto [D, FuD] = tf.dependent_async([]() {
  std::cout << "Task D\n";
}, B, C);
FuD.get();
```

Listing 2: A dynamic task graph programming example of Taskflow's AsyncTask interface [14].

Listing 2 presents a DTGP example of the same structure as Listing 1. Similarly, the program *explains itself*. The program creates a task graph of four tasks, `A`, `B`, `C`, and `D`. The dependency constraints state that task `A` runs before task `B` and task `C`, and task `D` runs after task `B` and task `C`. However, unlike Listing 1, when a task is created from `dependent_async`, its execution starts immediately. That is, the task execution can overlap with the task graph construction (see Figure 10). For large task graph, such as those of millions of tasks and dependencies, this overlap can bring certain performance advantage.

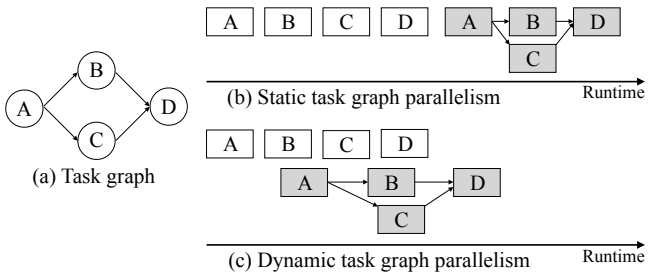

Figure 10: An illustration of the execution diagram of a task graph. White blocks denote the task creation and gray rectangles denote the task execution. Edges refer to the dependencies. (a) A task graph. (b) The execution diagram of STGP. (c) The execution diagram of DTGP.

6 CONCLUSION

In this paper, we have discussed several promising methods to accelerate STA by harnessing the power of CPU-GPU heterogeneous parallelism. We have discussed (1) task-parallel STA algorithms to improve timing propagation performance through asynchronous execution, (2) task graph partitioning algorithm to further improve the scheduling performance of task-parallel STA algorithms, (3) GPU kernel algorithms to largely speed up time-consuming STA tasks, and (4) a general-purpose task-parallel programming system that generalizes our approach to benefit other applications beyond STA. We hope this paper can inspire more solutions for enhancing the performance of STA.

ACKNOWLEDGMENTS

We are grateful for the support of four National Science Foundation (NSF) grants, CCF-2349141, CCF-2349582 (CAREER), OAC-2349143, and TI-2349144.

REFERENCES

[1] [n. d.]. DARPA IDEA Program. https://www.darpa.mil/news-events/2019-05-31
[2] [n. d.]. Intel TBB. https://github.com/01org/tbb
[3] [n. d.]. Motivo AI. https://motivo.ai/
[4] [n. d.]. OpenMP 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[5] [n. d.]. Qflow. http://opencircuitdesign.com/qflow/.
[6] OpenTimer. https://github.com/OpenTimer/OpenTimer
[7] Taskflow. https://github.com/taskflow/taskflow
[8] J. Bhasker et al. 2009. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer.
[9] Bérenger Bramas and Alain Ketterlin. 2020. Improving parallel executions by increasing task granularity in task-based runtime systems using acyclic DAG clustering. *PeerJ Computer Science* 6 (2020), e247.
[10] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE HPEC*. 1–7.
[11] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism Using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 283–284.
[12] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results. In *ACM/IEEE Design Automation Conference (DAC)*. 1388–1389.
[13] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2022. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *European Conference on Parallel Processing (Euro-Par)*. 468–479.
[14] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[15] Cheng-Hsiang Chiu, Zhicheng Xiong, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2024. An Efficient Task-parallel Pipeline Programming Framework. In *ACM International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*.
[16] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 923–929.
[17] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Degign Automation Conference (DAC)*. 1–6.
[18] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[19] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 721–726.
[20] Guannan Guo, Tsung-Wei Huang, and Martin Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1–6.
[21] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.
[22] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *ACM/IEEE Design Automation Conference (DAC)*. 715–720.
[23] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[24] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE TCAD* (2023).
[25] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE)*.
[26] Tsung-Wei Huang. 2020. A General-Purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.
[27] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–2.
[28] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 746–756.
[29] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 776–789.
[30] Tsung-Wei Huang and Leslie Hwang. 2022. Task-Parallel Programming with Constrained Parallelism. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.
[31] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2018. A General-Purpose Distributed Programming System Using Data-Parallel Streams. In *ACM International Conference on Multimedia (MM)*. 1360–1363.
[32] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 974–983.
[33] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2019. Essential Building Blocks for Creating an Open-Source EDA Project. In *ACM/IEEE DAC*. Article 78, 4 pages.
[34] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. 2017. DtCraft: A distributed execution engine for compute-intensive applications. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 757–765.
[35] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE DAC*. Article 229.
[36] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test* 38, 2 (2021), 62–68.
[37] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.
[38] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 41, 5 (2022), 1448–1452.
[39] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2021. Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40, 8 (2021), 1687–1700.
[40] Tsung-Wei Huang and Martin D. F. Wong. 2015. Accelerated Path-Based Timing Analysis with MapReduce. In *ACM International Symposium on Physical Design (ISPD)*. 103–110.
[41] Tsung-Wei Huang and Martin D. F. Wong. 2015. On fast timing closure: speeding up incremental path-based timing analysis with mapreduce. In *ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*. 1–6.
[42] Tsung-Wei Huang and Martin D. F. Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 895–902.
[43] Tsung-Wei Huang and Martin D. F. Wong. 2016. UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 35, 11 (2016), 1862–1875.
[44] Tsung-Wei Huang, Martin D. F. Wong, Debjit Sinha, Kerim Kalafala, and Natesan Venkateswaran. 2016. A distributed timing analysis framework for large designs. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[45] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 596–599.
[46] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 758–765.
[47] Tsung-Wei Huang Huang, Chun-Xun Lin, and Martin D. F. Wong. 2019. DtCraft: A High-Performance Distributed Execution Engine at Scale. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS (ICCAD)* 38, 6 (2019), 1070–1083.
[48] Shui Jiang, Tsung-Wei Huang, Bei Yu, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*.
[49] Shiu Jiang, Tsung-Wei Huangand, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE HPEC*. 1–7.
[50] B. Jin, G. Luo, and W. Zhang. 2016. A fast and accurate approach for common path pessimism removal in static timing analysis. In *IEEE ISCAS*. 2623–2626.
[51] Myung-Chul Kim et al. 2015. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In *IEEE/ACM ICCAD*. 921–926.
[52] Tin-Yin Lai, Tsung-Wei Huang, and Martin D. F. Wong. 2017. LibAbs: An Efficient and Accurate Timing Macro-Modeling Algorithm for Large Hierarchical Designs. In *ACM/IEEE Design Automation Conference (DAC)*.
[53] P. Lee, I. H. Jiang, and T. Chen. 2018. FastPass: Fast timing path search for generalized timing exception handling. In *IEEE/ACM ASPDAC*. 172–177.
[54] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM International Conference on Multimedia (MM)*. 2284–2287.

[55] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 64–71.

[56] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin D. F. Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI (GLVLSI)*. 183–188.

[57] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[58] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation Using Task Graph Parallelism. In *European Conference on Parallel Processing (Euro-Par)*.

[59] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 11 (2022), 3041–3052.

[60] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2023. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *International Conference on Parallel Processing (ICPP)*. 1–12.

[61] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*.

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

[63] Corentin Rossignon, Pascal Hénon, Olivier Aumage, and Samuel Thibault. 2013. A NUMA-aware fine grain parallelization framework for multi-core architecture. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1381–1390.

[64] Vivek Sarkar and John Hennessy. 1986. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 202–211.