G-PASTA: GPU-Accelerated Partitioning Algorithm for Static Timing Analysis

Boyang Zhang*
University of Wisconsin-Madison
bzhang523@wisc.edu

Dian-Lun Lin*
University of Wisconsin-Madison
dlin57@wisc.edu

Che Chang
University of Wisconsin-Madison
cchang289@wisc.edu

Cheng-Hsiang Chiu
University of Wisconsin-Madison
chenghsiang.chiu@wisc.edu

Bojue Wang
Rutgers University
bw391@math.Rutgers.edu

Wan Luan Lee
University of Wisconsin-Madison
wanluan.lee@wisc.edu

Chih-Chun Chang
University of Wisconsin-Madison
chih-chun.chang@wisc.edu

Donghao Fang
Texas A&M University
donghao@tamu.edu

Tsung-Wei Huang
University of Wisconsin-Madison
tsung-wei.huang@wisc.edu

ABSTRACT

Recent static timing analysis (STA) engines have leveraged task dependency graph (TDG) parallelism to accelerate various STA algorithms, including graph-based analysis and path-based analysis. Despite the promising speedup via task parallelism, the scheduling cost of a TDG has become dominant when handling large TDGs. To overcome this challenge, we propose G-PASTA, a simple and fast TDG partitioning algorithm to reduce the scheduling cost of large task-parallel STA algorithms. By harnessing the power of GPU computing, G-PASTA incurs minimal cost of partitioning while bringing significant runtime improvement to task-parallel STA algorithms. Compared to a state-of-the-art CPU-based TDG partitioner, G-PASTA is up to 41.8× faster in partitioning runtime and can improve the overall STA performance by 43% on large designs.

ACM Reference Format:
Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU-Accelerated Partitioning Algorithm for Static Timing Analysis. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656230

1 INTRODUCTION

Static timing analysis (STA) is an important stage in the overall design flow because it validates the expected timing behaviors of a circuit design. As the design complexity continues to grow, STA becomes very time-consuming. To alleviate the long runtime, recent STA tools, such as OpenTimer [4, 7] and many others [2, 3], have leveraged *task parallelism* to describe timing propagation algorithms in a top-down *task dependency graph* (TDG). Each TDG node represents a particular STA task (e.g., delay calculation, required arrival time update), and each TDG edge represents a dependency between two STA tasks. By delegating the scheduling of a TDG to a task execution environment, such as a dynamic scheduler [5, 6], we can efficiently parallelize timing propagation algorithms across manycore CPUs.

*Both authors contributed equally to this research.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656230

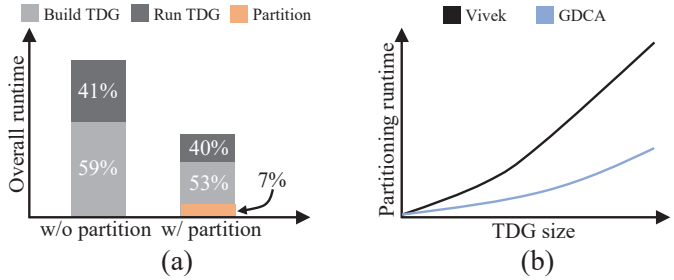

Figure 1: (a) Runtime breakdown of the core "update_timing" method in OpenTimer [4] with (right) and without (left) partitioning. (b) Growth of partitioning time with increasing TDG size for two popular TDG partitioners [1, 10].

Although TDG parallelism offers promising speedup, the scheduling cost–*comprising the building and execution of a TDG*–can become dominant when dealing with large task-parallel STA workloads [8]. For example, analyzing a circuit of 1.5M gates can spend over 50% runtime on building a TDG of 4M tasks and 5M dependencies, whereas the optimal execution performance is achievable using only 8–16 CPU threads [4]. This result implies that a large TDG is unnecessary given the small nubmer of saturated CPU threads. Furthermore, most timing propagation tasks exhibit relatively short runtime, comparable to or even shorter than per-task scheduling cost (i.e., assigning a task to a worker on a CPU core). For example, a backward propagation task in OpenTimer [4] takes about 0.5–50 us, while scheduling a task using OpenTimer's Taskflow scheduler [6] can take 0.2–3 us. Striking a balance between scheduling cost and task granularity is thus important for optimizing the performance of task-parallel STA algorithms.

A common solution for reducing scheduling cost is to break down a large TDG into many partitions, where each partition is a cluster of tasks that run sequentially with respect to their topological order in the original TDG. Instead of scheduling these clustered tasks one by one across different workers, we now only schedule a partition once and run it by a worker, which reduces the scheduling overhead. Note that TDG partitioning is very different from conventional graph or hypergraph partitioners (e.g., Metis [9] and Kahypar [11]) by focusing on partitioning a *directed acyclic graph* (DAG) to reduce the scheduling cost without affecting much the original TDG parallelism. Figure 1(a) shows the benefit of TDG partitioning by profiling the runtime of the core method "`update_timing`" in OpenTimer [4]. Building and running the original TDG take 59% and 41% of the runtime. After applying the proposed partitioner, despite incurring an extra cost for

partitioning, we can achieve nearly 50% runtime improvement due to reduced TDG size and scheduling cost.

There are a few popular TDG partitioners. Vivek [10] clusters tasks based on each task's impact on the overall TDG parallelism and critical path length. However, this clustering algorithm suffers from quadratic time complexity due to iterative checking of cycles. To solve this problem, GDCA [1] removes expensive cycle checking using breadth-first traversal, yet at the cost of reduced TDG parallelism. While these TDG partitioners help reduce the scheduling cost, they are all limited to *single-threaded* execution. As the TDG size increases, their partitioning time grows rapidly (see Figure 1(b)) and can outweigh the advantage of partitioning. To overcome this challenge, we propose *G-PASTA*, a fast GPU-powered TDG partitioning algorithm to improve the performance of task-parallel STA algorithms. We summarize three technical contributions of G-PASTA as follows:

- We design a GPU-accelerated partitioning algorithm that effectively partitions large TDGs into dependent subgraphs to reduce the scheduling cost.
- We design an efficient cycle-free clustering algorithm that can automatically cluster tasks to the right granularity without affecting too much the original TDG parallelism.
- We design a deterministic GPU kernel algorithm which allows applications to enable predictable results.

We evaluate the performance of G-PASTA on a set of large TDGs derived from a state-of-the-art task-parallel STA engine, OpenTimer [4]. Compared to GDCA, G-PASTA is up to 41.8× faster in partitioning runtime and can improve the overall STA performance by 43% on large designs.

2 PROBLEM DEFINITION AND CHALLENGES

Given (1) a TDG where each task represents a particular timing propagation task (e.g., required arrival time update) and each edge represents a task dependency and (2) a tunable parameter of partition size that restricts the maximum number of tasks per partition, our goal is to partition the TDG to the right granularity such that the partitioned TDG can produce the best runtime performance compared to the original TDG. Unlike the typical graph or hypergraph partitioning algorithms, TDG partitioning exhibits unique constraints and challenges, which we summarize below:

- Existing graph partitioning algorithms, such as Metis [9] and KaHypar[11] cannot be used out of the box for our application because their main goal is to minimize the cut size instead of the induced scheduling cost by a partitioning result. Furthermore, their solutions focus on balanced partitions instead of maximal parallelism which can significantly affect the overall runtime.
- Unlike undirected graph partitioning algorithms which may introduce cycles when clustering tasks together (see Figure 2(a)), a valid TDG partitioning result needs to be cycle-free (see Figure 2(b)). Otherwise, the result cannot be scheduled due to cyclic task dependencies.
- Partitioning a TDG in parallel often leads to non-deterministic outcomes that prevent applications from obtaining predictable results. For certain application scenarios (e.g., debugging), deterministic results are preferable. Thus, there is a requirement for an algorithm that allows applications to opt-in deterministic outcomes.

Furthermore, for many task-parallel STA workloads, the generated TDGs to perform parallel timing propagation can be huge (e.g., multi-millions of tasks and dependencies). To maximize the benefits of TDG partitioning, it is essential for the partitioning process to be as fast as possible. Otherwise, long partitioning time can outweigh the advantages gained from an improved TDG runtime. Compared to manycore CPUs, modern GPU offers order-of-magnitude more parallelism and memory bandwidth, which is particularly suitable for handling large volume of data. This advantage has inspired us to leverage the power of GPU computing to design a fast TDG partitioner targeting large task-parallel STA workloads.

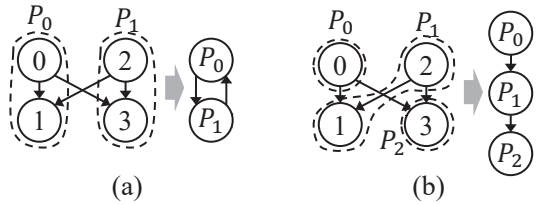

Figure 2: (a) An invalid TDG partitioning result due to cyclic dependencies between P0 and P1. (b) A valid TDG partitioning result where P0 includes task 0, P1 includes tasks 1 and 2, and P2 includes task 3.

3 G-PASTA

G-PASTA addresses the above challenges by introducing a parallelism-aware partitioning algorithm equipped with a cycle-free clustering method and a deterministic GPU kernel algorithm.

3.1 Parallelism-aware Partitioning Algorithm

Unlike the existing parallel graph partitioners that focus on minimizing the cut size of the partitioned graph, a parallel TDG partitioning algorithm focuses on reducing the scheduling overhead without affecting much the original TDG parallelism. Partitioning a large TDG in parallel requires efficient parallel traversal of the TDG, which can be achieved by the well-studied parallel breadth-first search (BFS). Based on BFS, the state-of-the-art GDCA obtains a levelized topological order of tasks and iteratively clusters tasks level by level to derive a partitioning result. However, this approach can largely reduce the TDG parallelism as nodes at the same level can run in parallel. As shown in Figure 3(a), GDCA can lead to a partitioning result of sequential execution.

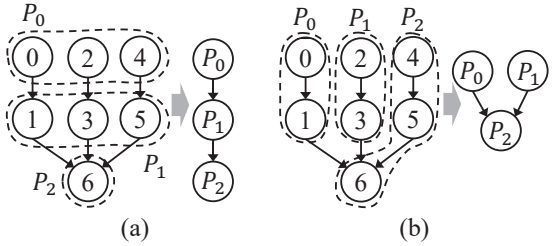

Figure 3: (a) Level-by-level partitioning method [1] can result in a sequential TDG. (b) Parallelism-aware partitioning method can produce a parallel TDG.

To overcome this challenge, G-PASTA introduces a parallelism-aware partitioning algorithm. Our algorithm prioritizes clustering tasks between adjacent levels to largely avoid reducing TDG parallelism. This is because tasks across different levels typically exhibit at least one dependency constraint. Compared to Figure 3(a), clustering tasks between adjacent levels as shown in (b) results in a smaller reduction of TDG parallelism and can thus produce a better partitioning result.

To this end, we design two arrays: (1) the desired partition ID (d_pid) array to store the IDs of the partition that each task desires to be clustered into, and (2) the final partition ID (f_pid) array to store

the IDs of the partition that each task is eventually clustered into. At each BFS level, the *f_pid* of each task is used to assign the *d_pid* of its neighbors in the next BFS level. Algorithm 1 presents G-PASTA's kernel for one partitioning iteration. To cluster the tasks between adjacent levels, our partitioning kernel consists of two steps as follows: *assign f_pid for current-level tasks by d_pid* and *assign d_pid and release neighboring dependencies*.

3.1.1 Assign f_pid for current-level tasks by d_pid. In step one, each thread handles one task in the current BFS level by grabbing a task from the array *handle*, which stores all the tasks to be handled in the current BFS level. Then each thread assigns the *f_id* of the task with the *d_pid* of the task, which is the *f_pid* of its parent task in the previous level. We use a partition size counter *pid_cnt* to count the number of tasks within a partition. If the desired partition of a task is not full (see line 5 of Algorithm 1, *Ps* is the partition size), we assign the *d_pid* of the task to the *f_pid* of the task. Otherwise, the task is assigned to a new partition.

3.1.2 Assign d_pid and release neighboring dependencies. In step two, each thread first assigns *d_pid* for the neighbors of the current task. Then, each thread releases the dependencies of the neighbors, i.e., marks the number of visited dependency edges. Note that the dependency mentioned in Algorithm 1 only refers to the fan-in dependency. We use a dependency counter *dep_cnt*, which is initialized as the number of dependencies of the neighbor, to record the remaining number of dependencies. If the neighbor task's dependents are fully released (i.e., *dep_cnt* = 0), this neighbor is pushed into *handle* for the next partitioning iteration.

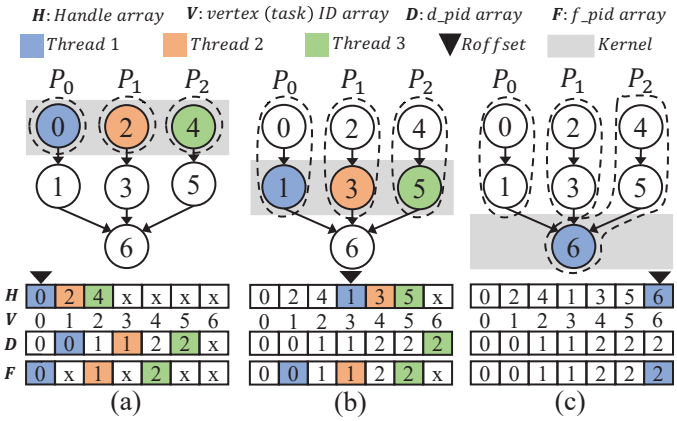

Figure 4: An example of our partitioning algorithm in three iterations under the partition size of 3. *H* is the array to store the tasks to be handled by threads. *V* is the array of vertex (task) IDs. *D* is the array of the desired partition ID for each task. *F* is the array of the final partition ID for each task.

To give a better understanding of how Algorithm 1 helps maintain the TDG parallelism during partitioning, we use Figure 4 to demonstrate our algorithm in three partitioning iterations under the partition size of 3. In Figure 4, *H* is the array to store the tasks to be handled by the threads. *V* is the array of vertex (task) ID. *D* is the array of the desired partition ID (*d_pid*) for each task. *F* is the array of the final partition ID (*f_pid*) for each task. The grey rectangle on the TDG refers to G-PASTA's partitioning kernel. The blue, orange, and green entries in the array refer to the entries that are handled by threads during one partitioning iteration. The black triangle is the reading offset (*Roffset*), which is an index in *H* to indicate the beginning of the task sequence in *H* to be handled by threads. We also define the reading size (*Rsize*) as the total number of tasks to be handled by threads during one partitioning kernel as shown in line 2 of Algorithm 1.

Before partitioning, all the source tasks (tasks 0, 2, 4) are pushed into *H*, and each source task is assigned with a different *d_pid* (0, 1, 2). *Roffset* is initialized as 0. We also initialize *Rsize* as the number of source tasks (3). Figure 4(a) shows the first partitioning iteration. G-PASTA's partitioning kernel is invoked at the first level of the TDG. With the *Roffset* as 0 and the *Rsize* as 3, threads 1, 2, and 3 handle tasks 0, 2, and 4, respectively. Based on the initialized *d_pid* of tasks 1, 2, and 3 in *D*, threads 1, 2, and 3 assign the *f_pid* for tasks 0, 2, and 4 as 0, 1, and 2 accordingly, marked as blue, orange, and green entries in *F*. Then, threads 1, 2, and 3 update the *d_pid* of the neighboring tasks of tasks 0, 2, and 4, which are tasks 1, 3, and 5, as 0, 1, and 2 separately, marked as blue, orange, and green entries in *D*. Since tasks 1, 3, and 5 have all their dependencies released by the threads, they are pushed into *H*. Then, we update the *Roffset* as 3 and the *Rsize* as 3 for the next iteration since three tasks are written into *H* in the first iteration.

Figure 4(b) shows the second partitioning iteration. The partitioning kernel moves to the second level of the TDG. With the *Roffset* as 3 and the *Rsize* as 3, threads 1, 2, and 3 handle tasks 1, 3, and 5. Based on the *d_pid* of tasks 1, 3, and 5, which are 0, 1, and 2, respectively, threads 1, 2, and 3 assign the *f_pid* of tasks 1, 3, and 5 as 0, 1, and 2 accordingly since the *pid_cnt* of P_0, P_1, and P_2 is less than the partition size (3). Then threads 1, 2, and 3 simultaneously try to assign the *d_pid* for task 6. Based on our proposed cycle-free clustering algorithm, which will be discussed in the next section, thread 3 eventually assigns the *d_pid* of task 6 as 2. The partitioning kernel finishes in Figure 4(c) by assigning the *f_pid* of task 6 as 2.

Algorithm 1: G-PASTA partitioning kernel

```
1:  /* Step 1: assign f_pid for current-level tasks by d_pid */
2:  parallel for each thread gid { /* gid < Rsize */
3:      cur = handle[Roffset + gid];
4:      cur_pid = d_pid[cur];
5:      if (atomicAdd(pid_cnt[cur_pid], 1) < Ps) then
6:          f_pid[cur] = cur_pid;
7:      else then
8:          new_pid = atomicAdd(max_pid, 1) + 1;
9:          f_pid[cur] = new_pid;
10:         pid_cnt[new_pid] ++;
11: }
12: /* Step 2: assign d_pid and release neighboring dependencies */
13: parallel for each thread gid {
14:     for each n ∈ neighbors of cur
15:         /* cycle_free_clustering_algorithm() */
16:         atomicMax(d_pid[n], f_pid[cur]);
17:         if (atomicSub(dep_cnt[n], 1) == 1) then
18:             Woffset = atomicAdd(Wsize, 1);
19:             handle[Roffset + Rsize + Woffset] = n;
20: }
```

3.2 Cycle-free Clustering Algorithm

To ensure the partitioned TDG is cycle-free, one simple solution is to iteratively check cycles and avoid clustering tasks that can introduce cyclic dependencies during the partitioning process [10]. However, iterative cycle checking can incur significant runtime overhead. To solve this problem, GDCA [1] partitions the TDG based on its topological order and clusters tasks level by level to avoid backward dependencies. Nevertheless, as shown in Figure 3(a), this method can largely reduce

the TDG parallelism because tasks at the same level should run in parallel. To overcome these challenges, we propose a simple yet efficient cycle-free clustering algorithm by restricting the parent partition to which a task can be clustered.

Specifically, we assign each task a partition ID. When multiple partitions want to cluster a task, only the partition with the *largest* ID can cluster this task. Since we traverse the TDG level by level using BFS, the partition IDs of all tasks at one level are always larger than those at previous levels. This organization implies no cyclic dependencies because the partition with a larger ID will always come after a partition with a smaller ID. Formally speaking, given a DAG, G, for each vertex $v_i \in G$, we define $pid(i)$ as the partition ID of v_i and $PRE(i)$ as the set of parent vertices of v_i. We denote P as a partition in G and $V(P)$ as the set of vertices within P. We say P is convex when (1) $\forall v_i, v_j \in V(P)$, $i \neq j$, and (2) $\forall v_k$ in any paths between v_i and v_j, $v_k \in V(P)$. With these notations, we outline our clustering rule as follows: $\forall v_i \in G$, $pid(i) = max\{pid(j) \mid v_j \in PRE(i)\}$.

Theorem 1. The proposed clustering algorithm does not introduce any cycle during the partitioning process.

Proof. We summarize three cases where a cycle can happen during the partitioning process and prove that none of them can exist in our algorithm. The first case where a cycle can occur is when P is not convex. As shown in Figure 5(a), P_0 is not convex as v_1 is in the path between v_0 and v_2 and $v_1 \notin P_0$. Based on our algorithm, we have $pid(0) < pid(1)$ and $pid(1) < pid(2)$. Thus $pid(0) < pid(2)$. However, this is contradictory to the fact that $pid(0) = pid(2)$ as $v_0, v_2 \in V(P_0)$. Thus, the first case will not exist and P must always be convex following our algorithm. The second case is when a cycle occurs among partitions even when partitions are all convex. As shown in Figure 5(b), assuming P_0 and P_1 are convex, there is a cycle between them. Based on our algorithm, we have $pid(0) < pid(1)$ and $pid(4) < pid(2)$. Besides, $pid(1) = pid(4)$ as $v_1, v_4 \in V(P_1)$. Thus $pid(0) < pid(2)$. However, this is contradictory to the fact that $pid(0) = pid(2)$ as $v_0, v_2 \in P_0$. Thus the second case won't exist. The case in Figure 5(b) can be extended to a more general case as shown in Figure 5(c), where there is a cycle among n convex partitions. Based on the above analysis, eventually we have $pid(0) < pid(n+2)$, which is still contradictory to the fact that $pid(0) = pid(n+2)$ as $v_0, v_{n+2} \in V(P_0)$. Thus, the more general case as shown in Figure 5(c) will not exist. □

The proposed cycle-free clustering algorithm can be efficiently implemented using just one lightweight GPU atomic operation, as shown in lines 15–16 in Algorithm 1. As a result, our algorithm is easy to implement and is extremely efficient with little kernel overhead. Besides, our algorithm always decides a lower bound for the number of partitions in the TDG since partitions with smaller IDs cannot continue to cluster tasks if all the available tasks are clustered by the partition with the largest ID. For instance, in Figure 4(c), there are no available tasks to cluster for P_0 and P_1. Thus, our algorithm guarantees a lower bound for the TDG parallelism regardless of the partition size. This property highlights the advantage of G-PASTA that users do not need to fine-tune a partition size as GDCA but simply use the original TDG size as the default value.

3.3 Deterministic GPU Kernel Algorithm

Although our cycle-free kernel algorithm is efficient in parallel clustering, it can introduce non-deterministic partitioning results, preventing certain applications of interest from obtaining predictable outcomes. As shown in Figure 6, two of the four tasks (task 0, 1, 2, and 3) can be clustered into P_0. However, the selection of which two tasks are clustered is entirely determined by the runtime. To further enhance our algorithm by offering an option for predictable partitioning results, we propose an efficient deterministic GPU kernel algorithm. As presented in Algorithm 2, our kernel algorithm consists of four steps: *sort the handle array and the desired partition ID array*, *identify the first task in each partition*, *determine if a desired partition is full*, and *assign deterministic partitioning results*.

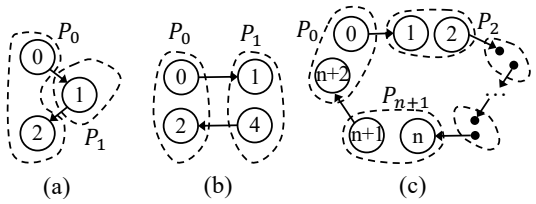

Figure 5: Three cases where cycles are introduced.

Algorithm 2: G-PASTA deterministic kernel

```
1:  /* Step 1: sort the handle arr and the desired partition ID arr */
2:  parallel for each thread gid {
3:      key_arr[gid] = d_pid[gid] « 32 | handle[gid];
4:  }
5:  handle = parallel_sort_by_key(handle, key);
6:  d_pid = get_d_pid_sort(handle, d_pid);
7:  /* Step 2: identify the first task in each partition */
8:  ones ← an array of ones; // reduce values
9:  fir_tid_arr = parallel_reduce_by_key(ones, d_pid);
10: fir_tid_arr = parallel_exclusive_scan(fir_tid_arr);
11: /* Step 3: determine if a desired partition is full */
12: parallel for each thread gid {
13:     fir_index = binarySearch(gid, fir_tid_arr);
14:     num_left = Ps - pid_cnt[handle[Roffset + gid]];
15:     if (gid < num_left + fir_tid_arr[fir_index]) then
16:         is_full_pid[gid] = 0;
17:     else then
18:         is_full_pid[gid] = 1;
19: }
20: num_full_arr = parallel_inclusive_scan(is_full_pid);
21: /* Step 4: assign deterministic partitioning results */
22: parallel for each thread gid {
23:     if (is_full_pid[gid] == 1) then
24:         f_pid[gid] = max_pid + num_full_arr[gid];
25:     else then
26:         f_pid[gid] = d_pid[gid];
27:     atomicAdd(pid_cnt[f_pid[gid]], 1);
28: }
29: max_pid += num_full_arr.back();
```

3.3.1 Sort the handle array and the desired partition ID array. The goal of this step is to sort the handle array and the desired partition ID array by each task's desired partition ID, such that tasks with the same desired partition ID are grouped together. As presented in lines 1–6 of Algorithm 2, we create a 64-bit sorting key array *key_arr* where the left 32 bits of each key store the desired partition ID, and the right 32 bits store the task ID of a given task. We then apply the `parallel_reduce_by_key` method on the handle array. This sort-by-key strategy eliminates the non-deterministic arising from the original ordering of tasks in the handle array. By the sorted handle array, we can also obtain the sorted desired partition ID array, *d_pid*. As shown in Figure 6, after step 1, tasks 0, 1, 2, and 3 with $d\_pid = 0$ are grouped together, and tasks 4 and 5 with $d\_pid = 1$ are grouped together.

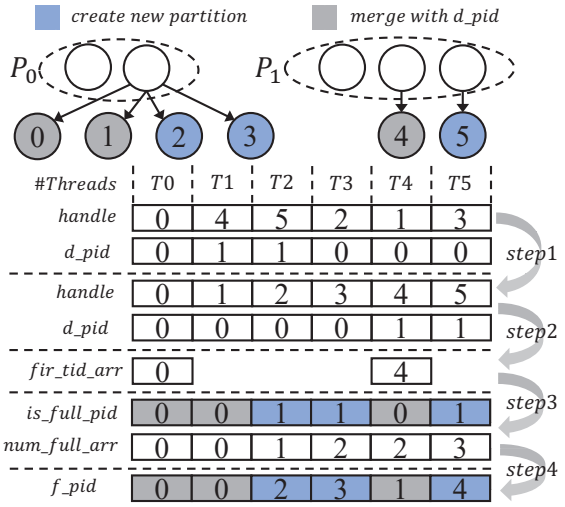

Figure 6: Example of the deterministic GPU kernel algorithm under the partition size of 4. A grey vertex represents a task that can be clustered into its desired partition. A blue vertex represents a task with a new partition.

3.3.2 Identify the first task in each partition. Based on the sorted arrays, the goal of this step is to identify the first task in each partition. We create fir_tid_arr to store the ID of each first task. As presented in lines 7–10 of Algorithm 2, to get fir_tid_arr, we apply `parallel_reduce_by_key` on an array of ones with the key as d_pid_sort to get the size of each partition. We then store the reduced values in fir_tid_arr and apply `parallel_exclusive_scan` on fir_tid_arr to obtain the final result. As shown in Figure 6, after step 2, f_tid has two entries where 0 represents that task 0 is the first task in P_0, and 4 represents that task 4 is the first task in P_1.

3.3.3 Determine if a desired partition is full. The goal of this step is to determine if a task can be clustered into its desired partition. If a task's desired partition is full, we need to create a new partition for that task and increment the maximum partition ID, max_pid. To this end, we create two arrays, is_full_arr and num_full_arr. is_full_arr indicates if each task's desired partition is full or not. num_full_arr indicates the accumulated number of full partitions corresponding to is_full_arr. Specifically, num_full_arr is the prefix sum (i.e., scan) of is_full_arr. As presented in lines 11–20 of Algorithm 2, we assign a task to a GPU thread. Each thread applies `binary_search` on the fir_tid_arr to find which desired partition the task belongs to. We then calculate num_left, the remaining number of tasks that can be clustered into that desired partition. If the task v can still be clustered into its desired partition, $is\_full\_arr[v] = 0$; otherwise, $is\_full\_arr[v] = 1$. After we obtain is_full_arr, we derive num_full_arr by applying `parallel_inclusive_scan` on the is_full_arr.

3.3.4 Assign deterministic partitioning results. Based on the is_full_arr and num_full_arr, we finally calculate the deterministic partitioning results and store them into f_pid. As presented in lines 21–29 of Algorithm 2, we assign a task to a GPU thread. We then check whether a task can be clustered into its desired partition using if_full_arr. If not, we assign the task to a new partition where the new partition ID is the maximum partition ID plus the task's corresponding element in num_full_arr. This organization not only ensures the assigned partition ID per task is deterministic, but also avoids synchronization that atomically gets a new partition ID for a task. Finally, we update pid_cnt for the next partition iteration. As shown in Figure 6, tasks 0, 1, and 4 are clustered into their desired partition (shown in grey), whereas tasks 2, 3, and 5 is 1 are assigned into new partitions (shown in blue). After four steps described in Algorithm 2, we increment the maximum partition ID by the number of new partitions created in this iteration (i.e., the last element in num_full_arr).

4 EXPERIMENTAL RESULTS

We implemented G-PASTA in C++ and CUDA and compiled it using nvcc v12.2 with `-O2` and `-std=c++17` enabled. We performed experiments on a 4.8 GHz 64-bit Linux machine equipped with an Intel Core i5-13500 CPU and an Nvidia RTX A4000 GPU. We compare the performance of G-PASTA with its two variants, seq-G-PASTA and deter-G-PASTA, and a baseline GDCA [1]. Seq-G-PASTA is a sequential, CPU-based implementation of G-PASTA using a single thread. Deter-G-PASTA incorporates the proposed deterministic GPU kernel algorithm to produce deterministic partitioning results.

We consider GDCA as our baseline due to its efficiency. As GDCA requires users to provide a partition size, we fine-tune it and use the value that produces the best performance for each circuit; for G-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 different partitioners into OpenTimer [4] and run graph-based analysis (`update_timing` command) on six industrial circuits. Specifically, when calling `update_timing`, OpenTimer will generate a TDG to perform parallel timing update. Statistics of these circuits and their generated TDGs are listed in Table 1. All data is an average of 10 runs.

4.1 Partition Performance Comparison

Table 1 compares the overall performance among GDCA, seq-G-PASTA, G-PASTA, and deter-G-PASTA in terms of their runtime improvement on 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 partitioners and their speedup over the baseline GDCA. We measure the performance in one full-timing iteration through the `update_timing` method in OpenTimer. The largest circuit, leon2, generates a TDG of 4.3M tasks and 5.3M dependencies to perform parallel timing update.

In general, all partitioners can improve the performance of `update_timing` due to reduced TDG size and scheduling cost. For instance, GDCA can improve the TDG runtime of the smallest circuit (aes_core) from 4.7 ms to 3.1 ms (1.5×) and of the largest circuit (leon2) from 349.1 ms to 193.5 ms (1.8×). Regardless of the improvement by GDCA, G-PASTA always outperforms GDCA. For instance, G-PASTA can improve the TDG runtime of the six circuits by 1.7–2.0×, whereas GDCA is 1.5–1.8×. Similar results can be observed in the other two variants of G-PASTA, seq-G-PASTA (1.7–2.0×) and deter-G-PASTA (1.7–2.0×). We attribute this result to G-PASTA's parallelism-aware partitioning algorithm that minimizes the impact on the original TDG parallelism during task clustering.

In terms of partitioning runtime, G-PASTA has demonstrated superior performance over GDCA because of the efficiency of our GPU kernel algorithm. The largest speedup values are observed in the three largest circuits, leon3mp, netcard, and leon2, where G-PASTA is 38.5×, 34.2×, and 41.8× faster than GDCA. On the other hand, we can see that deter-G-PASTA is a bit slower than G-PASTA (though still much faster than GDCA) due to the overhead of our deterministic kernel algorithm. Even without GPU, seq-G-PASTA is still 2.4–6.2× faster than GDCA across all circuits. As a result, we cannot see any benefit of GDCA because its long partitioning runtime outweighs its advantage in improved TDG runtime.

Table 1: Overall performance comparison among different partitioners (GDCA, seq-G-PASTA, G-PASTA, deter-G-PASTA) and their improvements on generated TDGs in the core `update_timing` method of OpenTimer [4].

circuit	#tasks	#deps	T_{TDG} (ms)	T_{TDG^P} (ms)				$T_{Partition}$ (ms)			
				GDCA	seq-G-PASTA	G-PASTA	deter-G-PASTA	GDCA	seq-G-PASTA	G-PASTA	deter-G-PASTA
aes_core	66.8K	86.4K	4.7	3.1 (1.5×)	2.3 (2.0×)	2.3 (2.0×)	2.4 (1.9×)	7.3	1.9 (3.8×)	2.3 (3.1×)	12.4
des_perf	303.7K	387.3K	25.5	16.0 (1.5×)	13.5 (1.8×)	13.6 (1.8×)	13.4 (1.9×)	49.4	9.4 (5.2×)	3.5 (14.1×)	16.4 (3.0×)
vga_lcd	397.8K	498.9K	33.5	21.2 (1.5×)	19.2 (1.7×)	19.3 (1.7×)	19.0 (1.7×)	70.7	11.4 (6.2×)	3.7 (19.1×)	17.1 (4.1×)
leon3mp	3.4M	4.1M	265.9	153.0 (1.7×)	131.8 (2.0×)	133.1 (1.9×)	130.8 (2.0×)	727.9	261.1 (2.7×)	18.9 (38.5×)	61.3 (11.8×)
netcard	4.0M	4.9M	312.1	175.2 (1.7×)	153.3 (2.0×)	154.7 (2.0×)	151.2 (2.0×)	856.8	338.8 (2.5×)	25.0 (34.2×)	61.1 (14.0×)
leon2	4.3M	5.3M	349.1	193.5 (1.8×)	173.3 (2.0×)	172.7 (2.0×)	171.1 (2.0×)	986.9	399.2 (2.4×)	23.6 (41.8×)	67.6 (14.5×)

T_{TDG}: TDG runtime before partitioning T_{TDG^P}: TDG runtime after partitioning $T_{Partition}$: partitioning runtime

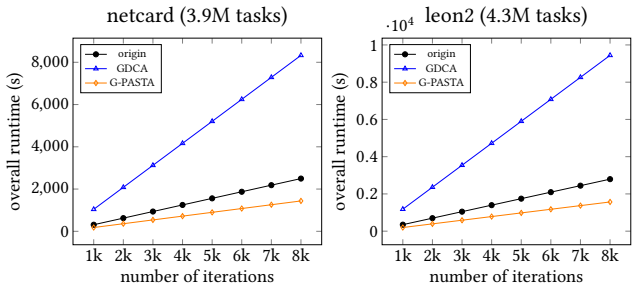

Figure 7: Comparison of STA runtime improvement between GDCA and G-PASTA over 8K incremental timing iterations. The black lines represents the original runtime without any partitioning.

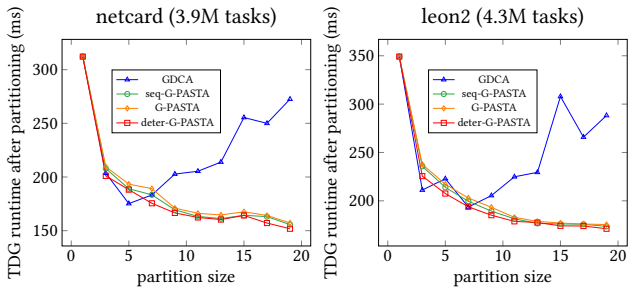

Figure 8: Comparison of TDG runtime (after partitioning) among GDCA, seq-G-PASTA, G-PASTA, and deter-G-PASTA under different partition sizes.

4.2 STA Runtime Comparison

Figure 7 compares the overall STA runtime between GDCA and G-PASTA over 8K incremental timing iterations, where the given partitioner is iteratively issued at each call to a design modifier followed by `update_timing`. The overall STA runtime includes the time of partitioning, construction, and execution of partitioned TDGs. The black line represents the original STA runtime without applying any partitioners to the generated TDGs. As we can see, G-PASTA largely improves the performance of `update_timing` due to its fast partitioning runtime and high partitioning quality. The improvement continues to accumulate as we increase the number of incremental timing iterations. However, we do not observe any benefit of using GDCA primarily because of its long partitioning runtime. For instance, running 8K iterations on leon2, G-PASTA speeds up the overall STA performance by 43%, whereas GDCA slows down the process by 3.7×.

4.3 TDG Runtime vs Different Partition Sizes

Figure 8 compares the TDG runtime (after partitioning) among GDCA, seq-G-PASTA, G-PASTA, and deter-G-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, it is user's responsibility to find the right partition size that produces the best performance. However, for G-PASTA, the TDG runtime continues to decrease until saturation (e.g., partition size of 15 for leon2). This is because Algorithm 1 always decides a lower bound for the number of partitions that cannot be clustered together, guaranteeing a lower bound for the resulting TDG parallelism. This property highlights another advantage of G-PASTA that user does not need to fine-tune the partition size but can simply use the original TDG size as the default value. G-PASTA will automatically converge to the right partition size and granularity that produce the best TDG runtime performance.

5 CONCLUSION

In this paper, we have proposed G-PASTA, a fast TDG partitioning algorithm to reduce the scheduling cost of large task-parallel STA algorithms. G-PASTA introduces an efficient cycle-free clustering algorithm that can automatically cluster tasks to the right granularity without affecting too much the TDG parallelism. Compared to a state-of-the-art CPU-based TDG partitioner, G-PASTA is up to 41.8× faster in partitioning runtime and can improve the overall STA performance by 43% on large designs.

ACKNOWLEDGMENT

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

REFERENCES

[1] 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.*
[2] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. Gpu-accelerated path-based timing analysis. In *2021 58th ACM/IEEE Design Automation Conference (DAC).* IEEE.
[3] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. Gpu-accelerated static timing analysis. In *ICCAD.*
[4] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD.*
[5] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++. In *IEEE IPDPS.* 974–983.
[6] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE TPDS.*
[7] Tsung-Wei Huang and Martin D. F. Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM ICCAD.* 895–902.
[8] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM ISPD.* 51–59.
[9] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.
[10] Vivek Sarkar and John Hennessy. 1986. Partitioning parallel programs for macro-dataflow. In *ACM LFP.*
[11] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics.*