G-STAR: GPU-Accelerated Statistical Static Timing Analysis Using Level-by-Level Replication

Boyang Zhang, Chih-Chun Chang, Yi-Hua Chung, Che Chang, Cheng-Hsiang Chiu, Aditya Das Sarma, and Tsung-Wei Huang(✉)

University of Wisconsin-Madison, Madison, WI, USA
{bzhang523,chih-chun.chang,yihua.chung,cchang289,chenghsiang.chiu, adassarma,tsung-wei.huang}@wisc.edu

Abstract. Statistical static timing analysis (SSTA) is a critical component of modern Electronic Design Automation (EDA) flows for modeling on-chip process variations. Among various SSTA methods, *first-order block-based SSTA* (FB-SSTA) supports efficient level-by-level timing propagation with high data parallelism, making it particularly suitable for GPU acceleration. However, existing GPU-based FB-SSTA solutions assume that the entire timing graph and data fit in GPU memory, an assumption that breaks down as circuit size increases. To overcome this challenge, we propose G-STAR, a GPU-accelerated FB-SSTA algorithm designed for memory-constrained FB-SSTA workloads. G-STAR enables efficient levelized FB-SSTA via level-by-level replication, combined with a replicated-pin scheduler that reuses GPU memory and avoids redundant phase propagation. Compared to a state-of-the-art GPU solution, G-STAR achieves up to 9.24× speedup on large designs and successfully completes FB-SSTA on multi-million-pin circuits where the baseline fails due to GPU *out-of-memory* errors.

1 Introduction

Statistical static timing analysis (SSTA) is a critical stage in Electronic Design Automation (EDA). Compared to traditional static timing analysis (STA), it enables more accurate delay estimation by modeling on-chip process variation (OCV) as random variables [3]. Among existing SSTA solutions, *Monte Carlo-based* SSTA (MC-SSTA) has been widely explored. For example, [8] introduces an MC-SSTA framework that uses QMC sampling and Karhunen-Loéve expansion to reduce dimensionality. Although MC-SSTA achieves high accuracy, the reliance on extensive sampling often leads to very long runtime (e.g., hours for million-gate designs [7]), limiting its practical use in industrial settings.

To overcome this runtime limitation, [9] introduces *first-order block-based SSTA* (FB-SSTA), a *level-by-level* timing-propagation method that approximates timing distributions by linearly propagating statistical moments. Unlike MC-SSTA, FB-SSTA computes first-order sensitivities of all timing quantities to

M. Torquati et al. (Eds.): Euro-Par 2026, LNCS 16781, pp. 553–567, 2027.
https://doi.org/10.1007/978-3-032-35248-4_38

each variation source, so it does not require any time-consuming sampling processes. Figure 1(a) illustrates the FB-SSTA process on a circuit of two gates [7]. In the circuit, each pin stores multiple *phases* of timing quantities, such as delay (d), voltage fluctuation (ΔV), temperature shift (ΔT), and others projected to one PVT (Process, Voltage, Temperature) condition. In this example, we show three phases, so each pin stores three phases of timing quantities. In real designs, there can be hundreds or even thousands of phases [4]. To perform FB-SSTA on this circuit, a common solution is to construct its level list, as shown in Fig. 1(b), and then propagate the timing quantities level by level from primary input (PI) pins to primary output (PO) pins through the circuit network.

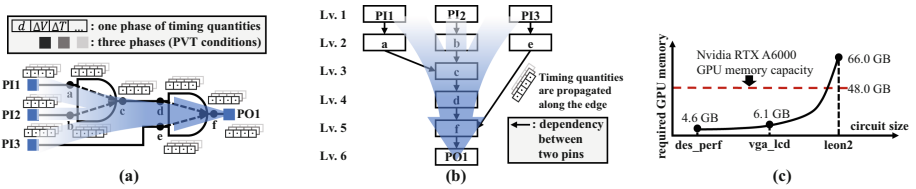

Fig. 1. (a) Illustration of first-order block-based SSTA (FB-SSTA) on a circuit of two gates. (b) Timing quantities are propagated level by level from primary inputs (PIs) to primary outputs (POs). (c) Required GPU memory of existing level-by-level FB-SSTA algorithms can exceed the available capacity when the phase count is large (e.g., 1024).

The level-by-level propagation in FB-SSTA, together with the massive data parallelism that appears as circuits grow in size and complexity, makes FB-SSTA well-suited for GPU acceleration. To this end, [6] presents a GPU-accelerated FB-SSTA framework that parallelizes levelized timing propagation while explicitly handling spatial correlations through principal component analysis. More recently, [5] proposes a GPU-accelerated FB-SSTA engine that models statistical timing propagation in a differentiable form to support downstream physical design optimization. Despite runtime improvement via GPU, nearly all existing GPU-based FB-SSTA solutions assume that the entire timing graph and its data, i.e., the level list and phase data in Fig. 1(b) fit in GPU memory. In other words, they overlook a practical challenge, one we encountered when working with a major EDA vendor, the ever-increasing phase complexity and circuit size cannot easily be accommodated within GPU memory. For example, as shown in Fig. 1(c), under 1024 phases, performing level-by-level FB-SSTA on leon2 (4.3M pins) requires 66.0 GB of GPU memory. This exceeds the 48.0 GB available on Nvidia RTX A6000, which is already large for modern GPUs.

To address this memory challenge, [1] proposes G-STP, a GPU-accelerated FB-SSTA algorithm designed specifically for memory-constrained SSTA workloads. G-STP models an FB-SSTA workload as a *task graph*, where each task represents the propagation of timing quantities at a pin and each edge represents the dependency between two tasks. It then pre-allocates the maximum available GPU memory and uses a job queue to schedule these propagation tasks pin

by pin. This scheduling process is GPU memory–aware coupled with a heuristic to minimize the GPU memory footprint during execution. However, G-STP has two major drawbacks. First, it relies on a job queue to decide which pin to process next, which can introduce significant CPU scheduling overhead as the circuit size grows. For example, under a circuit with 397.8K pins, this overhead can reach 26.46% of the total runtime (discussed later in Fig. 7). Second, based on its scheduling strategy, G-STP launches kernels and data transfers pin by pin, leading to excessive kernel and data transfer launches when the circuit size grows. Since each launch only handles the timing propagation of a single pin, these small, frequent launches underutilize the GPU's parallel resources and become highly inefficient.

Consequently, we propose G-STAR, a GPU-accelerated SSTA engine that performs fast levelized timing propagation through a level-by-level replication strategy. To enable correct levelization of timing quantities under limited GPU memory, G-STAR inserts replicated pins on selected levels. It also introduces a replicated-pin scheduler that reuses GPU memory locations to reduce memory usage after replication, together with an efficient GPU kernel that minimizes unnecessary computation and memory accesses on replicated pins. We evaluate G-STAR on ten industrial circuits generated by OpenTimer and compare it with G-STP. Experimental results show that G-STAR achieves up to 9.24× speedup in overall SSTA performance on large designs, and successfully completes SSTA on several circuits with more than 3M pins where G-STP fails due to GPU out-of-memory (OOM) errors.

2 Background

2.1 Problem Formulation

G-STAR is inspired by the memory-constrained SSTA problem introduced by G-STP [1], as well as our collaboration with a major EDA vendor to retarget their FB-SSTA tool to GPUs. For simplicity, we hereafter use the term SSTA to denote the FB-SSTA paradigm. As illustrated in Fig. 2, consider the two-phase case: in (a), each circuit pin stores two phases of timing quantities, where each phase contains one min delay (d^{min}) and one max delay (d^{max}). On the GPU, these delays are organized into a min phase vector (P_{min}) and a max phase vector (P_{max}), as shown in (b). The length of both P_{min} and P_{max} equals the phase count, which is two in this example. We refer to these two vectors collectively as *phase data*. With these organizations, the goal is to propagate phase data from PIs to POs of the circuit, a process we refer to as *phase propagation*. Phase propagation becomes more expensive as the number of phases increases, since each pin must store a longer phase vector in memory. In practice, the phase count often reaches the thousands, resulting in substantial storage demands [4,7]. Following the G-STP formulation and current industrial practice [1], we focus on circuits whose phase data fit in CPU memory but cannot be fully accommodated in GPU memory as the phase count increases. Although CUDA Unified Memory could be used in this case, it is not ideal for high-performance SSTA kernels

with predictable access patterns, since automatic CPU–GPU migration may trigger unnecessary page faults and degrade performance. Therefore, we manage memory transfers explicitly to maintain predictable performance.

Figure 2(c) illustrates the phase propagation process from pins $PI1$ and $PI2$ to pin c through gate $AND1$. Here, P^imin and P^imax represent the min and max phase vectors of pin i. The phase vectors from $PI1$ and $PI2$ first propagate to a and b by adding the edge delays (d_{edge}), and are then combined at $AND1$ using `min` and `max` operations. These operations are guided by a per-gate permutation vector (π), whose length equals the phase count (two in this example). In SSTA, π aligns random-variable coefficients so timing operations can be computed correctly [1]. It defines how the input phase vectors are combined to produce the output phase vectors, as described in Eq. 1. If a gate has multiple outputs, the same permutation vector is shared across all outputs.

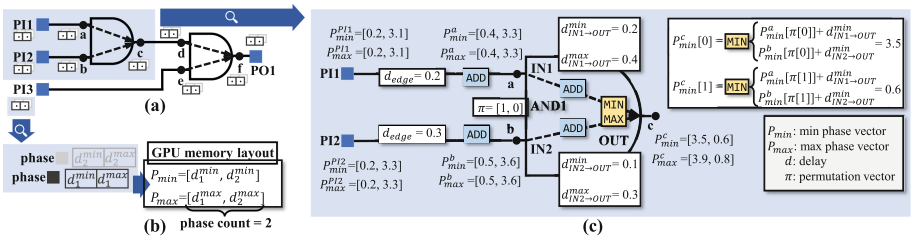

Fig. 2. (a) Illustration of SSTA on the circuit in Fig. 1. (b) Under two phases, each pin carries two sets containing one min delay (d^{min}) and one max delay (d^{max}), arranged in one min phase vector (P_{min}) and one max phase vector (P_{max}) on GPU. (c) Phase vector (P_{min} and P_{max}) propagation from $PI1$ and $PI2$ to pin c through gate $AND1$.

$$P^c_{min}[i] = \texttt{min}(P^a_{min}[\pi[i]] + d^{min}_{IN1\to OUT}, P^b_{min}[\pi[i]] + d^{min}_{IN2\to OUT}) \tag{1}$$

Equation 1 shows how the min phase vector of pin c (P^c_{min}) is computed. For each entry i in P^c_{min}, we use the permutation vector π to select the corresponding entries from P^a_{min} and P^b_{min}. We then add the appropriate edge delays and take the `min` of the two resulting values. This produces the result for entry i in P^c_{min}. The max phase vector of c (P^c_{max}) is computed in the same way, except that the `min` operation is replaced with `max`.

2.2 Limitations of State-of-the-Art Methods

Recent GPU-accelerated SSTA frameworks [2,5,6] improve performance by exploiting the large amount of data parallelism in SSTA, but they assume that the entire timing graph and all phase data fit in GPU memory. This assumption becomes impractical as circuit size and phase count grow. To address this issue, G-STP [1] introduces a GPU-parallel approach for memory-constrained

SSTA workloads. As shown in Fig. 3(a), G-STP models phase propagation as a task graph, where each task performs phase propagation for a single pin, including CPU-GPU data transfers and GPU kernel execution. During execution, the scheduler maintains a job queue to decide which pin to process next, and transfers only the required phase data into pre-allocated GPU memory, as illustrated in Fig. 3(b). This pin-by-pin scheduling strategy reduces GPU memory usage through a memory-aware scheduling heuristic.

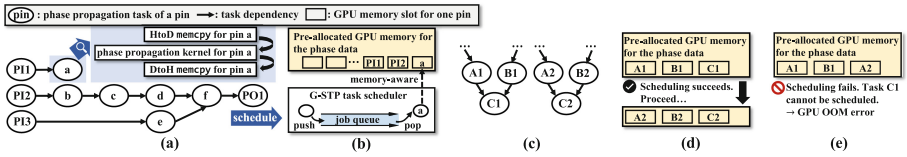

Fig. 3. (a)–(b) illustrate SSTA in G-STP. (a) The task graph representation of the circuit in Fig. 1. (b) The task graph is scheduled on G-STP's task scheduler to perform SSTA. (c)–(e) demonstrate the OOM issue in G-STP. (c) An example task graph assuming the GPU can store phase data for three pins. (d) A successful scheduling of the task graph. (e) A failed scheduling where G-STP encounters a GPU OOM error.

Although G-STP can perform SSTA under limited GPU memory, its pin-by-pin scheduling strategy introduces two major drawbacks. First, it incurs high CPU scheduling overhead and many small GPU operations, including frequent kernel launches and `memcpy`s, which reduces GPU utilization. Second, the scheduler can still fail with GPU out-of-memory (OOM) errors because task execution depends on the order in which ready tasks are selected. This behavior is shown in Fig. 3(c)–(e). In this example task graph (c), we assume that after tasks A1 and B1 are scheduled, both tasks C1 and A2 become ready. In (d), if the scheduler happens to pick C1 next, then tasks A1, B1, and C1 can finish and free their memory slots for the remaining tasks, and the schedule succeeds. However, in (e), if the scheduler instead picks A2 first, then A2 occupies the memory slot that C1 needs. Once A2's phase data are loaded, all three memory slots become full, leaving no free slot for C1's phase data. Because C1 cannot be scheduled for phase propagation, G-STP reaches a dead end and encounters a GPU OOM error, meaning the GPU has no available memory to load the phase data for the next required pin.

To reduce CPU scheduling overhead and better utilize GPU parallelism, many existing works [5–7,9] use *levelized* phase propagation, which processes phase data level by level in batches. However, this approach becomes increasingly limited by GPU memory capacity as circuit size and phase count grow. Existing implementations often store the entire levelized structure and all associated phase data on the GPU, which can exceed available GPU memory for large circuits with millions of pins, as illustrated in Fig. 1(c).

3 G-STAR

As a consequence, we introduce G-STAR, a GPU-accelerated SSTA algorithm based on the levelized approach. To enable levelized phase propagation under the GPU memory constraint, we first identify the minimum amount of phase data that needs to be stored on the GPU at once. In levelized propagation, phase data can be propagated between adjacent levels as long as no multi-level (i.e., > 2) dependencies exist. In this scenario, we try to store only two consecutive levels of phase data, a previous level (*prev*) and a current level (*cur*), to reduce the GPU memory requirement, as shown in Fig. 4. Specifically, we pre-allocate one *prev* buffer and one *cur* buffer on the GPU, each large enough to hold the phase data of the largest level in the circuit (three pins in this example). During phase propagation, we first transfer the phase data of the previous level from the CPU to *prev*, and then launch a kernel that propagates the entire level of phase data from *prev* to *cur*. After the kernel finishes, we transfer the results in *cur* back to the CPU. Finally, we perform a pointer swap between *prev* and *cur* so that, in the next level, the kernel can again read from *prev* and write to *cur* without extra `memcpy` overhead. This full process is illustrated in Fig. 4(b).

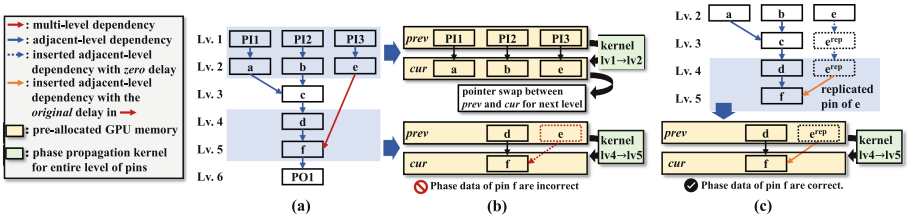

Fig. 4. Illustration of levelized phase propagation on GPU when only two consecutive levels (*prev* and *cur*) of phase data are stored. (a) Level list from Fig. 1. (b) Incorrect propagation caused by a multi-level dependency. (c) We replicate pin e twice to remove its multi-level dependency so that the phase data propagate correctly from e to f.

However, while this method only requires storing two consecutive levels of phase data, it can produce incorrect phase results, as illustrated in Fig. 4(b). After the kernel propagates the phase data from level 4 to level 5, the phase data of pin f are incorrect because the phase data of its dependent, pin e, never get propagated to level 4 due to the multi-level dependency between e and f.

To address this problem, the key idea behind G-STAR is to reconstruct the level list by inserting replicated pins in selected levels to replace all multi-level dependencies with adjacent-level dependencies. In G-STAR, we first apply a level-by-level replication algorithm to insert replicated pins where necessary, ensuring that phase data are propagated correctly. We then apply a replicated-pin scheduler to reuse GPU memory locations for replicated pins and reduce the memory overhead introduced by replication. Finally, G-STAR performs levelized phase propagation with an efficient GPU kernel.

Algorithm 1: Level-by-level replication

```
1  // L: original level list
2  // pq: fanout pins ordered by level (descending)
3  foreach l ∈ L do
4  |  foreach cur_p ∈ l do
5  |  |  push fanouts with multi-level dependencies into pq;
6  |  |  f_largest ← pq.pop();
7  |  |  insert replicated pins between cur_p and f_largest;
8  |  |  add adjacent-level dependencies (original + zero delay);
9  |  |  while pq not empty
10 |  |  |  f ← pq.pop(); connect f to its replicated pin with original delay;
```

3.1 Level-by-Level Replication Algorithm

G-STAR replaces all multi-level dependencies with adjacent-level dependencies in the level list through a level-by-level replication algorithm, as shown in Fig. 4(c). To remove the multi-level dependency from pin e to pin f (marked in red in (a)), we insert two replicated pins of e in levels 3 and 4 and connect them with adjacent-level dependencies. Because phase data must accumulate edge delays as they propagate, the final inserted dependency from the replicated pin in level 4 to f (marked in orange) must preserve the *original delay* of the multi-level dependency it replaces, while the remaining inserted dependencies (dashed) have zero delay. With these replicated pins in place, the phase data of e are correctly propagated to level 4 and appear in *prev*, allowing us to compute the phase data of f correctly.

Algorithm 1 details this strategy. For each pin cur_p in each level, we examine its fanout pins and push those with multi-level dependencies into a priority queue ordered by descending pin level (lines 3–5). We then pop the fanout pin with the largest level $f_{largest}$, and insert the necessary replicated pins between cur_p and $f_{largest}$, adding adjacent-level dependencies with the appropriate original or zero delay (lines 6–8). Next, we pop the remaining fanout pins from the queue and link each to its corresponding replicated pin using one dependency edge that preserves the original delay, without introducing extra replicated pins.

3.2 Replicated-Pin Scheduler

Although the level-by-level replication algorithm enables correct phase propagation, it can still introduce a large number of replicated pins into the reconstructed level list. This is because modern circuits contain many multi-level dependencies, and these dependencies often span many levels. To replace such long-range dependencies with adjacent-level dependencies, a pin may need to be replicated multiple times across multiple intermediate levels. For example, as illustrated in Fig. 5(a)–(b), removing only three multi-level dependencies requires adding eight replicated pins, even though the original level list contains only seven pins. To mitigate this issue, we design a replicated-pin scheduler for G-STAR that

Algorithm 2: Replicated-pin scheduler

```
1  // lifetime_rep: a vector of [p_rep, first_level, last_level]
2  // slots: a list of [slot_index, free_level]
3  foreach p_rep ∈ L_rep
4  |  [first_level, last_level] ← get_lifetime(p_rep);
5  |  lifetime_rep.push(p_rep, first_level, last_level);
6  sort lifetime_rep by first_level (ascending);
7  foreach [p_rep, first_level, last_level] ∈ lifetime_rep
8  |  assign ← false;
9  |  foreach [slot_index, free_level] ∈ slots
10 |  |  if free_level ≤ first_level
11 |  |  |  p_rep.slot_index ← slot_index;
12 |  |  |  free_level ← last_level + 2;
13 |  |  |  assign ← true;    break;
14 |  if lassign
15 |  |  slot_index_new ← slots.size();
16 |  |  free_level_new ← last_level + 2;
17 |  |  p_rep.slot_index ← slot_index_new;
18 |  |  slots.push(slot_index_new, free_level_new);
```

reuses GPU memory locations (*slots*) to reduce the memory requirement after replication.

As part of this design, we pre-allocate a standalone *rep* buffer on the GPU to store phase data exclusively for replicated pins, and use the scheduler to assign each replicated pin to a memory slot in *rep* for its phase data. Because the phase data of replicated pins reside entirely in *rep*, they are reused across levels without being transferred back to the CPU, avoiding additional CPU–GPU data transfer overhead. Algorithm 2 details this scheduler design. The scheduler first computes the lifetime of each replicated pin p_{rep}, defined by the first and last levels at which the pin appears (lines 3–5). A replicated pin's phase data are written only once at its first level (*first_level*) and read only once at its last level (*last_level*). Within this lifetime, no additional phase computation or data transfer is required. The scheduler then sorts replicated pins by their *first_level* and assigns them to slots in *rep* using a greedy reuse strategy: if an existing slot becomes free before this pin's lifetime begins, that slot is reused (lines 9–13); otherwise, a new slot is allocated (lines 14–18). To ensure correctness, a slot is marked as free only after *last_level* + 2 rather than *last_level* + 1. This additional one-level delay prevents the slot from being reused while the phase data of the replicated pin are still being read at *last_level*, a situation that can lead to incorrect results.

Figure 5(a)–(b) illustrates Algorithm 2, with explanation outlined in the caption. By reusing slots whose phase data are no longer needed, the scheduler minimizes the number of slots required in *rep* while maintaining correctness. More importantly, this clear lifetime management of replicated-pin phase data enables an efficient kernel design. In the next section, we show how this design avoids

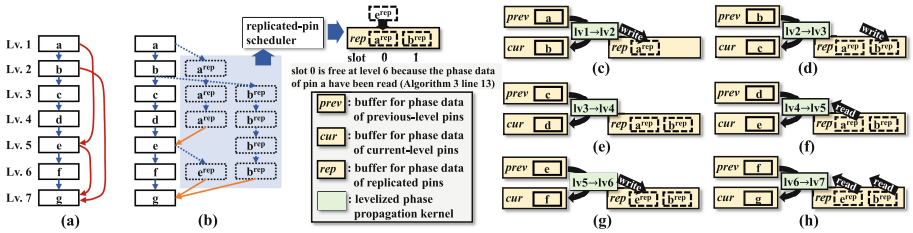

Fig. 5. (a)–(b) Example of applying the replicated-pin scheduler to a level list, where eight replicated pins are added to remove three multi-level dependencies (marked in red). The scheduler reuses memory slots in *rep*, allowing the phase data of all eight replicated pins to be stored using only two slots. Because the last level of replicated pin a^{rep} is 4, its slot is released at level 6, allowing replicated pin e^{rep} to reuse slot 0 (Algorithm 2, line 13). (c)–(h) Levelized execution of the phase propagation kernel guided by the replicated-pin scheduler.

redundant phase propagation for replicated pins during levelized execution. It is also worth noting that both Algorithm 1 and Algorithm 2 are performed only once during initialization before SSTA execution. The reconstructed level list can then be reused for all subsequent SSTA iterations, making the replication cost a one-time preprocessing overhead.

3.3 Levelized Phase Propagation Kernel

Building on the replicated-pin scheduler (Algorithm 2), we design a levelized phase propagation kernel that avoids redundant phase propagation for replicated pins. Algorithm 3 presents the kernel design. In this kernel, the phase data of each pin are handled by T GPU threads, where each thread handles a subset of phases with a stride size of T. For each pin, the kernel first checks whether it is a replicated pin. If the pin is replicated and its phase data have already been written to the *rep* buffer in its first level, the kernel immediately skips further computation for that pin (lines 4–6). This early-exit mechanism ensures that each replicated pin's phase data are written to *rep* only once, thereby avoiding redundant phase propagation across levels. After this check, the kernel determines the destination buffer for storing the results: original pins write their phase data to *cur*, while replicated pins write to their assigned slots in *rep*. During phase propagation, the kernel reads input phase data from either *prev* or *rep*, depending on whether a fanin pin is original or replicated. For gate output pins, the kernel aggregates phase data across all fanins using `min` and `max` operations (lines 11–20). For non-gate pins, which have only a single fanin, the kernel directly propagates the phase data by adding the corresponding delay (lines 21–25). Figure 5(c)–(h) visualizes this levelized kernel execution process.

With the replicated-pin scheduler, the kernel knows exactly when to access each replicated pin's phase data, eliminating unnecessary accesses. More importantly, since the kernel executes level by level, the scheduling overhead is bounded

Algorithm 3: Levelized phase propagation kernel

```
1  // p: pin (id) handled by this thread; phase_offset: first phase handled by this
   thread (stride = T); T: number of threads per pin; π: permutation vector;
   N_phases: phase count; d: edge delay; src_min/max, dst_min/max: input/output
   phase data (min/max delays)
2  parallel for each thread tid
3    p ← tid / T;
4    if p.is_replicated and p.first_level < current level
5      // skip replicated pins already written to rep
6      return;
7    phase_offset ← tid % T;
8    π ← p.gate_permutation;
9    dst ← p.is_replicated ? rep : cur;
10   for phase ← phase_offset to N_phases − 1 step T
11     if p.is_gate_output_pin
12       min_val ← +∞;   max_val ← −∞;
13       foreach fanin pin f ∈ p.fanins
14         src ← fanin_p.is_replicated ? rep : prev;
15         min_tmp ← src_min[f][π[phase]] + d_min^fanin;
16         max_tmp ← src_max[f][π[phase]] + d_max^fanin;
17         min_val ← min(min_val, min_tmp);
18         max_val ← max(max_val, max_tmp);
19       dst_min[p][phase] ← min_val;
20       dst_max[p][phase] ← max_val;
21     else
22       f ← p.fanin;
23       src ← fanin_p.is_replicated ? rep : prev;
24       dst_min[p][phase] ← src_min[f][phase]+d_min^fanin;
25       dst_max[p][phase] ← src_max[f][phase]+d_max^fanin;
```

by the maximum number of levels rather than the size of the circuit graph, as in G-STP.

4 Experimental Results

We implemented G-STAR in C++ and CUDA and compiled it using nvcc v12.2 with -O2 and -std=c++17 enabled. We performed experiments on a 2.0 GHz 64-bit Linux machine equipped with an Intel Xeon Gold 6138 CPU with 182 GB RAM, and an Nvidia Tesla V100 GPU of 32 GB global memory. We consider G-STP [1] as our baseline since it represents the state-of-the-art GPU-accelerated SSTA algorithm designed for memory-constrained SSTA workload. We compare the performance of G-STAR and G-STP on ten industrial circuits. These circuits, generated using the widely-used open-source timer OpenTimer [4], are configured by default with 1024 phases to accommodate various statistical corners. Note that 1024 phases is also the configuration used to generate the golden

reference for result validation. Later, we will evaluate G-STAR's performance under different phase counts. Circuit statistics are given in Table 1, where #pins and #edges only count the pins and edges in the original circuit before adding replicated pins, and each result is the average of 10 independent runs.

4.1 Overall Performance Comparison

Table 1 compares the overall performance between G-STP and G-STAR. The value under the mem_{CPU} column shows the total CPU memory required to store all phases of the entire circuit. Note that mem_{CPU} includes only the phases of original pins. The phases of replicated pins are stored in the GPU and managed by the replicated-pin scheduler, which will be discussed in Section ??. The value under the $\text{mem}_{GPU}^{G-STAR}$ column represents the GPU memory required by G-STAR to perform correct SSTA, including the memory required to store phases for original pins (mem_{orig}) and additional memory required for replicated pins (mem_{rep}). We do not report the required GPU memory for G-STP because its allocation is dynamic and depends on the available memory at runtime.

In terms of total SSTA runtime, G-STAR consistently outperforms G-STP across all circuits. For small circuits with 13.1K to 66.8K pins, G-STAR achieves 7.06×–8.30× speedups over G-STP. For medium circuits with 303.7K to 397.8K pins, the speedups further increase to 8.62×–9.24×, with the highest value observed on des_perf. For large circuits with 2.4M to 4.3M pins, the advantages of G-STAR become even more pronounced. Specifically, G-STAR can successfully complete all large circuits, while G-STP fails on many (leon3mp, netcard, and leon2) due to out-of-memory (OOM) errors. The only large circuit that G-STP can complete is usb_phy_ispd, for which G-STAR is 8.41× faster.

We attribute the performance advantage of G-STAR over G-STP to our efficient levelized phase propagation, which avoids the complex task graph–based propagation used in G-STP. This levelized approach minimizes CPU-side overhead during SSTA and supports more efficient data transfers. Further details will be discussed in Sect. 4.3.

4.2 Performance Under Different Phase Counts

Figure 6 shows the speedup of G-STAR over G-STP under different phase counts for des_perf and vga_lcd. We focus on these two circuits because they are the largest ones that both G-STP and G-STAR can complete within our chosen range of phase counts. Overall, the speedup of G-STAR over G-STP gradually decreases as the phase count increases, since larger phase counts result in longer kernel execution times and reduce the relative contribution of CPU scheduling. The highest speedup appears at 256 phases, where G-STAR achieves 19.89× on des_perf and 22.48× on vga_lcd over G-STP. This trend is expected because G-STP performs kernel execution and memory transfers pin by pin, while G-STAR operates level by level. As the phase count increases, the pin-by-pin execution strategy in G-STP becomes relatively more efficient because each pin carries a larger workload. However, for larger circuits, such as leon3mp, netcard, and

Table 1. Comparison of overall performance between G-STP [1] and G-STAR on ten industrial circuits generated by OpenTimer [4] under 1024 phases.

circuit	#pins	#edges	mem_{CPU} (GB)	$\text{mem}_{GPU}^{G-STAR}$ (GB)		total SSTA runtime (ms)	
				mem_{orig}	mem_{rep}	G-STP	G-STAR
wb_dma	13.1K	16.5K	0.2	0.01	0.02	236.26	31.62 (7.47×)
tv80	17.0K	23.0K	0.26	0.01	0.04	265.15	37.51 (7.06×)
ac97_ctrl	42.4K	53.5K	0.65	0.03	0.07	660.09	84.72 (7.79×)
aes_core	66.8K	86.4K	1.02	0.05	0.12	1078.77	129.84 (8.30×)
des_perf	303.7K	387.3K	4.63	0.2	0.44	5367.41	580.50 (9.24×)
vga_lcd	397.8K	489.9K	6.07	0.5	0.88	6483.45	752.08 (8.62×)
usb_phy_ispd	2.4M	2.9M	37.33	3.08	4.65	41848.44	4973.78 (8.41×)
leon3mp	3.4M	4.1M	51.52	1.23	5.32	OOM	6901.37
netcard	4.0M	4.9M	61.02	2.59	7.2	OOM	7833.23
leon2	4.3M	5.3M	66.04	3.65	7.08	OOM	8499.19

#pins, #edges: total number of pins and edges in the original circuit before adding replicated pins; mem_{CPU}: total CPU memory to store all phases of the entire circuit (without replicated pins); mem_{orig}, mem_{rep}: required GPU memory in G-STAR to store phases for original and replicated pins; OOM: GPU out-of-memory error

leon2, G-STP cannot exploit this advantage because its algorithm encounters GPU OOM failures.

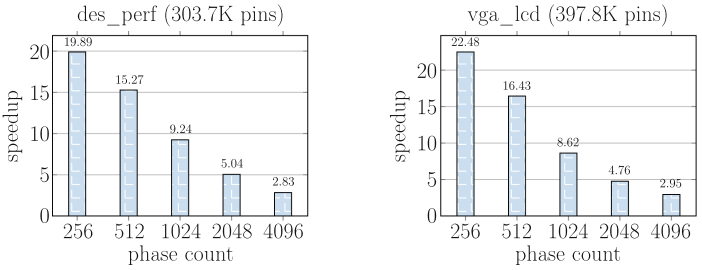

Fig. 6. Speedup of G-STAR over G-STP under different phase counts.

4.3 Runtime Breakdown

In this section, we first break down the total SSTA runtime of G-STAR and G-STP into two parts: *CPU scheduling* and *GPU execution*. CPU scheduling includes kernel launches, data transfer (`memcpy`) launches, and task-graph scheduling (used only in G-STP). GPU execution includes kernel execution and data transfers, including host-to-device (HtoD) and device-to-host (DtoH) `memcpy`. We then examine the GPU execution time in percentage terms.

Total SSTA Runtime Breakdown. Figure 7(a)–(b) show the total SSTA runtime breakdown in terms of CPU scheduling and GPU execution for vga_lcd under 4096 phases. We choose vga_lcd for this experiment because it is the largest one that can support 4096 phases without overflowing the CPU memory on our machine. In G-STP, GPU execution accounts for 73.54% of the total runtime, leaving a substantial 26.46% spent on CPU scheduling. In contrast, G-STAR achieves nearly 100% GPU execution time with only minimal CPU scheduling overhead. The higher CPU scheduling cost in G-STP comes from its task-graph execution model, which relies on a job queue to decide which pin to process next during SSTA execution and launches kernels and `memcpy` operations on a per-pin basis. These runtime scheduling decisions and frequent launches introduce significant CPU overhead. In contrast, G-STAR adopts a level-by-level replication strategy that enables a simpler and more efficient levelized phase-propagation model without an expensive runtime scheduler involved. The replication process (Algorithms 1 and 2) is performed once before SSTA execution and reused across iterations, incurring little CPU scheduling overhead. Moreover, the levelized model allows kernels and `memcpy` operations to be launched on a per-level basis, largely reducing the number of launches compared to G-STP.

GPU Runtime Breakdown. Figure 7(c)–(d) compares the GPU runtime breakdown of G-STP and G-STAR. A key difference lies in kernel execution time: in G-STP, kernels take 2003.5 ms (29.99%), whereas in G-STAR they take only 97.5 ms (3.01%). This reflects the higher kernel efficiency enabled by G-STAR's per-level kernel launches, compared to the per-pin launches used in G-STP. Despite this improvement, GPU runtime in both methods is still dominated by data transfer, accounting for 70.01% of the GPU runtime in G-STP and 96.99% in G-STAR. These transfers include both HtoD and DtoH `memcpy`. HtoD `memcpy` occurs only during initialization to transfer metadata (e.g., circuit topology) and input phase data to the GPU. As a result, most of the transfer time comes from DtoH `memcpy`, which moves the computed phase results back to the CPU and accounts for more than 99% of the total transfer time in this benchmark. However, this HtoD `memcpy` is necessary because the resulting phase data must be returned to the CPU for downstream SSTA applications to use, such as timing-driven optimization [7]. Note that the phase data of replicated pins do not contribute to this overhead, since they remain in the GPU *rep* buffer and are reused across levels without being transferred back to the CPU.

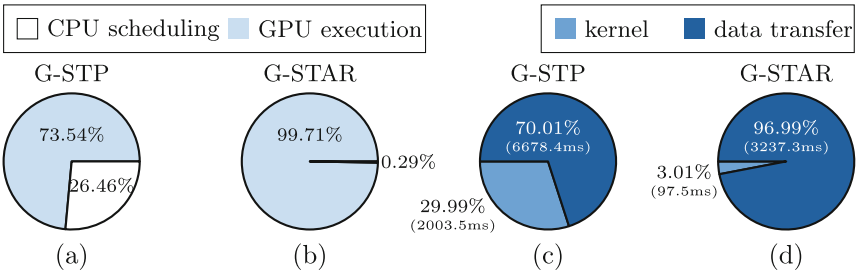

Fig. 7. Runtime breakdown comparison between G-STP and G-STAR for vga_lcd with 4096 cases. (a)–(b) show the total runtime breakdown. (c)–(d) show the GPU runtime breakdown with the actual runtime values shown below the percentages. Data transfer includes both HtoD and DtoH `memcpy` operations.

5 Conclusion

In this paper, we have proposed G-STAR, a GPU-accelerated FB-SSTA engine designed for memory-constrained FB-SSTA workloads. G-STAR enables efficient levelized FB-SSTA via level-by-level replication, combined with a replicated-pin scheduler that reuses GPU memory and an efficient kernel that avoids redundant phase propagation. Compared to a state-of-the-art GPU solution, G-STAR achieves up to 9.24× speedup on large designs and successfully completes FB-SSTA on multi-million-pin circuits where the baseline fails due to GPU OOM errors. This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. The authors would like to also thank reviewers for their constructive feedback in improving this paper.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Chang, C.C., Huang, T.W.: Statistical timing graph scheduling algorithm for GPU computation. In: ACM/IEEE Design Automation Conference (DAC) (2025)
2. Chiu, C.H., Morchdi, C., Chang, C.C., Yu, C., Zhou, Y., Huang, T.W.: Optimizing CUDA graph scheduling with reinforcement learning-a case study in SSTA propagation. In: 2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD), pp. 1–8. IEEE (2025)
3. Forzan, C., Pandini, D.: Statistical static timing analysis: a survey. Integration **42**(3), 409–435 (2009)
4. Huang, T.W., Guo, G., Lin, C.X., Wong, M.D.: Opentimer v2: a new parallel incremental timing analysis engine. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **40**(4), 776–789 (2020)
5. Lu, Y.C., Guo, Z., Kunal, K., Liang, R., Ren, H.: Insta: an ultra-fast, differentiable, statistical static timing analysis engine for industrial physical design applications.

In: 2025 62nd ACM/IEEE Design Automation Conference (DAC), pp. 1–7. IEEE (2025)
6. Shen, Y., Hu, J.: GPU acceleration for PCA-based statistical static timing analysis. In: 33rd IEEE International Conference on Computer Design (ICCD), pp. 674–679. IEEE (2015)
7. Sinha, D., Guerra e Silva, L., Wang, J., Raghunathan, S., Netrabile, D., Shebaita, A.: Tau 2013 variation aware timing analysis contest. In: Proceedings of the 2013 ACM International Symposium on Physical Design, pp. 171–178 (2013)
8. Veetil, V., Sylvester, D., Blaauw, D.: Efficient Monte Carlo based incremental statistical timing analysis. In: Proceedings of the 45th Annual Design Automation Conference, pp. 676–681 (2008)
9. Visweswariah, C., Ravindran, K., Kalafala, K., Walker, S.G., Narayan, S.: First-order incremental block-based statistical timing analysis. In: Proceedings of the 41st Annual Design Automation Conference, pp. 331–336 (2004)