HeteroLatch: A CPU-GPU Heterogeneous Latch-Aware Timing Analysis Engine

Xizhe Shi[1†], Zizheng Guo[1,2†], Yibo Lin[1,2,3*], Zuodong Zhang[2], Yun Liang[1,2,3], Runsheng Wang[1,2,3*]
[1]School of Integrated Circuits, Peking University [2]Institute of Electronic Design Automation, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits
Email: xizheshi@stu.pku.edu.cn, {gzz, yibolin, ericlyun, r.wang}@pku.edu.cn, zhangzd@pkueda.org.cn

***Abstract*—Latches, prevalent in high-frequency circuits, challenge timing analysis due to time borrowing and latch loops, complicating static timing analysis (STA) algorithms and parallelization strategies. To address these issues, we propose HeteroLatch, a CPU-GPU heterogeneous framework that enables efficient latch-aware timing analysis. By integrating adaptive loop handling with hierarchical parallel timing propagation, our method mitigates sequential bottlenecks through CPU-GPU collaboration, hiding graph decomposition overhead via early termination, while optimizing GPU throughput with dynamic workload allocation. Experimental results show average speed-ups of 12.64×, 9.45×, and 1.96× over industrial timers PrimeTime, OpenSTA, and SOTA work, respectively. HeteroLatch bridges the gap between latch-specific timing complexities and GPU acceleration, offering a scalable solution for advanced-node verification.**

I. Introduction

As the backbone of signoff and timing closure, static timing analysis (STA) plays a central role in design verification. STA engines are repeatedly invoked across design stages, requiring efficient support for diverse sequential elements and adaptability to increasing complexity and shrinking timelines [1]. These demands are especially acute in latch-intensive designs.

Unlike flip-flops, latches support time borrowing, which redistributes slack across cycles to ease timing pressure, and they also offer a simpler structure that saves area and power. However, latch flexibility introduces two core challenges: 1) time borrowing leads to clock-phase-dependent AT propagation, complicating setup checks and AT updates; 2) latch loops create cyclic dependencies that invalidate DAG-based routines.

Current industrial STA tools, such as PrimeTime [2] and OpenSTA [3], leverage clock-phase-aware modeling and iterative propagation to handle latch-based timing. As shown in Figure 1(a), latch D-Q arcs are disabled after delay computation to break loops and enable pin-based levelization. After AT propagation, setup checks at latch endpoints detect timing violations and capture time-borrowing behaviors. The extracted time-borrowing information is then used to guide AT updates through the disabled D-Q arcs and downstream paths, and the process repeats until all ATs converge. This modeling captures time borrowing through latch transparency windows rather than clock edges. For latch loops, timers apply iterative arc relaxations and repeated AT updates to ensure convergence.

† Equal contribution, * Corresponding authors.

This project is supported in part by the National Science Foundation of China (Grant No. T2293701), the Natural Science Foundation of Beijing, China (Grant No. Z230002), and the 111 Project (B18001).

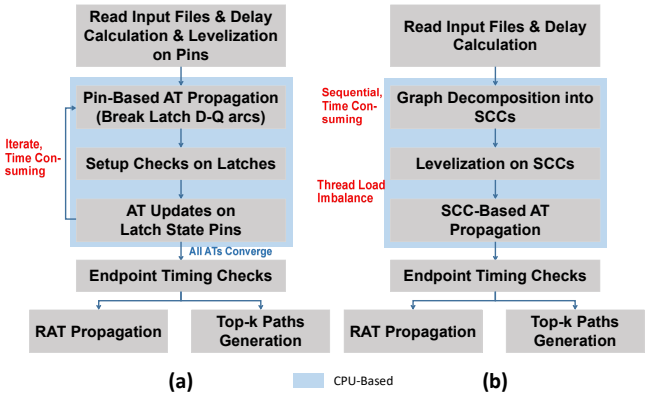

Fig. 1: Current frameworks of latch timing analysis in (a) industrial timers and (b) academic method in [4].

Despite their wide adoption, these approaches face notable limitations. Iterative updates from time borrowing induce redundant recomputation, propagating AT changes to subcircuits multiple times. Moreover, handling latch loops with global iteration strategies leads to worst-case $O(n^2)$ complexity in circuits with n pins. These inefficiencies are exacerbated in modern designs with dense latch clusters and deep latch loops.

Academic research has actively addressed latch-related STA challenges. For time-borrowing modeling, [5, 6] employed iterative and constraint-based strategies to handle latch transparency. For latch loop handling, early work [7] applied graph decomposition and traversal heuristics at the cost of precision. Later, [8] improved accuracy through iterative propagation, but incurred higher costs due to extra graph constructions. To accelerate latch analysis without accuracy loss, [4] proposed an SCC-based propagation framework that restructured timing graphs into strongly connected components (SCCs). As shown in Figure 1(b), this approach avoids cyclic dependencies via SCC-based propagation. However, it introduces scalability bottlenecks: SCC decomposition is sequential and resists parallelization; inclusion of latch D-Q arcs imposes extra dependencies; and variations in SCC sizes lead to significant workload imbalance between threads. These limitations hinder parallelism and scalability in latch-heavy designs.

Beyond sequential element handling, STA efficiency remains vital for iterative VLSI design closure. Heterogeneous CPU-GPU architectures have shown significant speedups in STA tasks such as delay characterization [9–11], graph analysis [12–15], and path-based evaluation [16–21]. However, these frameworks merely focus on flip-flop-based designs

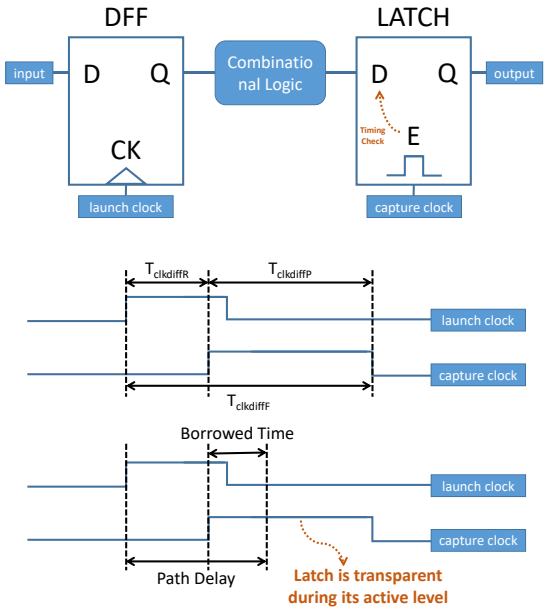

Fig. 2: A two-stage sequential circuit demonstrating time borrowing through latch transparency during its active phase.

and lack support for phase-sensitive constraints in latch-based circuits, limiting their applicability to advanced-node designs.

To address these limitations and enable latch-aware acceleration, we propose **HeteroLatch**, a CPU-GPU heterogeneous framework for STA that integrates the first GPU-accelerated engine that supports timing analysis on latch-based circuits. The design achieves this through three key contributions:

1) We propose a hybrid graph processing strategy for latch timing analysis. The novel framework divides AT propagation into two stages, resolving excessive iterations and limited parallelism in traditional algorithms.
2) Our CPU-GPU collaboration framework introduces a scheduling strategy that almost hides the overhead of sequential graph decomposition via possible early termination, leading to measurable acceleration in runtime.
3) We enable selective GPU parallelism with mixed granularity for AT propagation, dynamically applying thread- and block-level strategies to optimize runtime efficiency.

We integrate our algorithm into an STA engine and show up to 12.64×, 9.45×, and 1.96× speed-ups over PrimeTime, OpenSTA, and SOTA work [4], respectively on large designs.

II. Preliminaries

A. Time Borrowing and Latch Loops

The setup check mechanism for latches differs fundamentally from that of flip-flops due to their time-borrowing capability, where the data signal leverages the transparency window to "borrow" time from subsequent paths. In a two-stage circuit composed of a D flip-flop and an active-high transparent latch (Figure 2), the setup checks of the latch endpoint include: (1) aligning launch and capture clocks; and (2) computing the timing metrics *slack*, *actual_time_borrow* (ATB) and

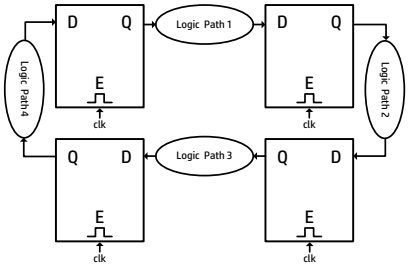

Fig. 3: An example of a latch loop.

max_time_borrow (MTB) to quantify transparency usage. They are calculated as follows:

$$MTB = T_{clkdiffP} - T_{setup}$$

$$Slack_{nominal} = T_{clkdiffR} - T_{setup} + AT_E - AT_D$$

$$ATB = \begin{cases} 0, & Slack_{nominal} \geq 0 \\ min(|Slack_{nominal}|, MTB), & Slack_{nominal} < 0 \end{cases}$$

$$Slack_{eff} = \begin{cases} Slack_{nominal}, & ATB = 0 \\ Slack_{nominal} + ATB, & 0 < ATB \leq MTB \\ Slack_{nominal} + MTB, & ATB > MTB \end{cases}$$

Time borrowing allows delayed data in the current cycle to be captured during the next cycle's transparency window. This resolves setup timing violations by reallocating unused slack from the subsequent cycle, without extending clock periods.

Notably, time borrowing remains inapplicable to hold time checks of latches, which require data stability after the clock edge and differ merely in clock alignment.

The AT_Q update, particularly the latch D-Q arc relaxation process (which differs from conventional arc relaxation), inherently involves multi-clock-domain timing behavior due to the transparency characteristic of latches. Its computation follows:

$$AT_Q = max(AT_D + Delay_{D-Q} + ATB, AT_E + Delay_{E-Q})$$

The introduction of latches can also generate latch loops in circuits (Figure 3), where combinational elements and latches form cyclic topologies. These loops disrupt DAG-based STA methodologies. Cyclic dependencies prevent pin levelization and force timing information such as ATs to propagate iteratively until convergence or divergence is detected.

TABLE I: Overview of latch-analysis support in different academic and commercial STA engines.

STA engine	Time borrowing analysis	Latch loop handling	Time complexity	GPU support
OpenTimer [22]	×	×	—	×
[23, 24]	✓	✓	$O(n^2)$	×
[4]	✓	✓	$O(\sum k_i^2)$	×
GPU Timers†	×	×	—	✓
OpenSTA [3]	✓	✓	$O(n^2)$	×
PrimeTime [2]	✓	✓	$O(n^2)$	×
This work	✓	✓	$O(\sum k_i^2)$	✓

n denotes pin num in circuits, k_i represents the pin num of each SCC.
†: GPU timer works include: [10, 11, 13–18, 20]. None of them supports timing analysis on latches.

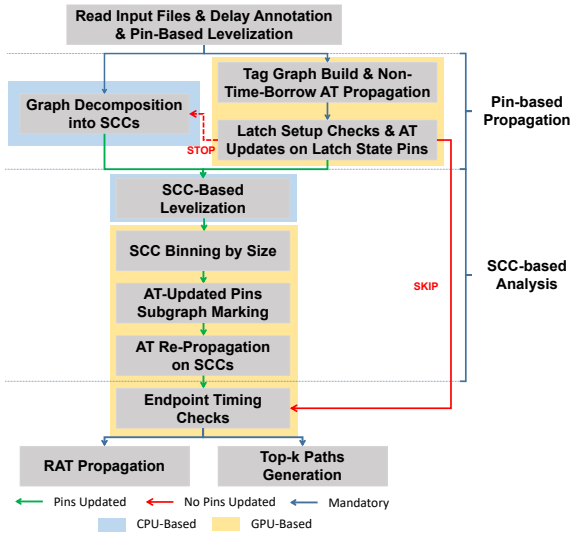

Fig. 4: Overall flow of HeteroLatch.

B. Latch Analysis with CPU-GPU Parallelism

As shown in Table I, existing industrial STA engines and [4] support latch analysis with time borrowing and loop handling, but remain CPU-bound with limited scalability beyond 8–16 cores. In contrast, existing GPU-based STA tools offer acceleration but lack support for latch-specific features, leaving a clear gap for a unified, GPU-accelerated latch timing engine.

However, designing GPU-accelerated latch-aware STA faces three core challenges: First, time-borrowing analysis mismatches GPU STA frameworks for flip-flop circuits, and iterative methods remain ill-suited to GPU architectures. Second, latch loop resolution imposes a trade-off: pin-level methods require many iterations, while coarse SCC-based propagation limits parallelism. They are both antithetical to GPU acceleration paradigms. Third, when adopting SCC-based AT propagation strategies, CPU-only graph decomposition nullifies GPU gains, as inherently sequential steps dominate runtime.

III. Algorithms

To address the limitations mentioned above and enable compatibility with GPU-accelerated frameworks, we propose **HeteroLatch**, an efficient heterogeneous STA engine for latch timing analysis. We build our flow on top of prior works including levelization on pins and SCCs, graph-based STA algorithms [4, 13, 14] and path search algorithms [16–18].

As shown in Figure 4, our framework divides full AT propagation into two stages: pin-based pre-propagation without time borrowing and on-demand SCC-based re-propagation, assisted by necessary auxiliary procedures. Latch setup checks are performed during these steps, while the final endpoint checks handle remaining constraints, such as latch hold checks.

We introduce four main parts of the flow: hybrid CPU-GPU preliminary timing analysis, SCC binning by size, subgraph marking, and SCC AT re-propagation. These components are mapped to CPU and GPU to efficiently handle latch behaviors.

Algorithm 1: Hybrid-CPU-GPU-Update-Timing

Input : Timing Graph G, Constraints Graph C
Output: Arrival Times (AT_{min}, AT_{max}), Actual Time Borrows ATB, SCCs S, Updated Pins Collection P

1 **initialize** $stop\_flag$ ▷ Thread-shared termination flag
2 **begin** Parallel processing between T1 and T2
3 **Thread T1: AT Pre-Propagation** ▷ GPU
4 build_tag_graphs(G, C)
5 $(AT_{min}, AT_{max}) \leftarrow$ propagate_at(G)
6 $ATB \leftarrow$ eval_latch_setup(AT_{min}, AT_{max}, C)
7 $P \leftarrow$ latch_at_reupdate$(G, \&AT_{max}, ATB)$
8 **if** $P = \emptyset$ **then** atomicSet$(stop_flag,$ **true**$)$
9 **Thread T2: Graph Decomposition** ▷ CPU
10 **for** *each unvisited pin* v *in* G **do**
11 Tarjan_SCC$(v, G, \&S, \&stop_flag)$
12 **synchronize** T1 and T2

Algorithm 2: Tarjan-SCC-with-Early-Termination

Input : Pin v, Timing Graph G, SCCs $\&S$
if is_set$(stop_flag)$ **then return** ▷ Early termination
$index \leftarrow index + 1$
$dfn[v] \leftarrow index$, $low[v] \leftarrow index$, push$(stack, v)$
for *each* v*'s fanout arc* $e(v, u)$ **do**
 if u *is not Visited* **then**
  call Tarjan_SCC(u) and mark u as *Visited*
  $low[v] \leftarrow min(low[v], low[u])$
 else if u *is in stack* **then**
  $low[v] \leftarrow min(low[v], dfn[u])$
if $low[v] = dfn[v]$ **then**
 push all pins in $stack$ into S as an SCC [25]

A. Hybrid CPU-GPU Preliminary Timing Analysis

To enable SCC-based re-propagation, we first apply Tarjan's algorithm [25] to decompose the graph. Although linear in complexity, this process is sequential and must run on a single CPU thread. Conversely, the AT pre-propagation stage, including tag graph construction, non-time-borrow AT propagation (with latch D-Q arcs disabled), latch setup checks and AT re-updates of latch state pins, exhibits high GPU parallelism. Since these two stages are data-independent, they can execute concurrently on CPU and GPU, as shown in Algorithm 1.

Based on whether latch state pin ATs are re-updated on the GPU and the relative runtime of GPU thread and CPU thread, our hybrid CPU-GPU timing analysis yields four cases:

Case 1. CPU Early Termination: CPU should have dominated runtime, but no AT_{max} re-updates on latch state pins ($P = \emptyset$) trigger CPU termination via $stop\_flag$. GPU sets the shared flag, and CPU's recursive Tarjan algorithm (Algorithm 2) detects the flag at each call entry and immediately returns, aborting entire graph decomposition. The algorithm skips the SCC stage and proceeds to endpoint timing checks.

Algorithm 3: SCC-Binning-by-Size

Input : SCC Levelization L
Output: SCC Size Boundary (for each level) SB

Call SCC_Binning_Kernel on L and get SB
Kernel Function SCC_Binning_Kernel:
  Input : SCC Levelization L
  Output: SCC Size Boundary (for each level) SB
  $levelID \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
  if $levelID \geq$ size(L) **then return**
  Use the two-pointer partitioning technique [26] to move all single-pin SCCs in $L[levelID]$ before multi-pin SCCs. Record the split position as $SB[levelID]$.

Algorithm 4: Subgraph-Marking-on-SCCs

Input : SCCs S, Updated Pins Collection P, Timing Graph G, SCC Levelization L

GPU Parallel for *each* p *in* P **do** ▷ Initial marking
  mark $S[scc\_id[p]]$ as *Updating*
for *each* l *in* L **do** ▷ Marks propagation level by level
  GPU Parallel for *each SCC* s *in* l **do**
    if s *is Updating* **then continue**
    for *each* p *in* s **do**
      for *each* p*'s fanin arc* $e(from, p)$ *in* G **do**
        if $scc\_id[p] = scc\_id[from]$ **then continue**
        if $S[scc\_id[from]]$ *is Updating* **then**
          mark $S[scc\_id[p]]$ as *Updating*

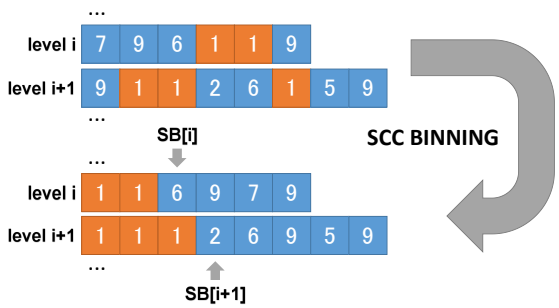

Fig. 5: Example of SCC Binning with SCC sizes in boxes.

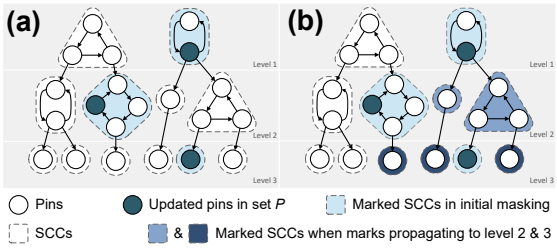

Fig. 6: Example of subgraph SCC marking.
(a) Phase 1: Mark SCCs that the updated pins belongs to.
(b) Phase 2: Marks propagate to level 2 & 3.

Case 2. CPU Runtime Hiding and SCC Stage Skipping: AT pre-propagation on GPU dominates runtime, fully hiding CPU overhead. With no AT_{max} re-updates on latch state pins ($P = \emptyset$), the SCC-based analysis stage is safely skipped.

Case 3. CPU Runtime Hiding with SCC Stage Invoked: GPU process dominates runtime and fully hides CPU overhead. However, AT_{max} re-updates ($P \neq \emptyset$) require re-propagation on the affected subgraph, triggering SCC-based analysis.

Case 4. Dynamic Sync: CPU process dominates runtime, and partial AT_{max} changes ($P \neq \emptyset$) require post-task synchronization. The GPU waits for the CPU to finish, after which the algorithm proceeds to the SCC-based analysis stage.

This design guarantees that graph decomposition overhead on CPU, which is traditionally a critical bottleneck, is either fully masked by GPU process and early termination (Cases 1-3) or partially mitigated through hybrid execution (Case 4).

B. SCC Binning by Size and Subgraph Marking

Graph decomposition converts the original cyclic graph into an SCC-based DAG. To start SCC-based analysis, preparatory steps are required: SCC levelization [4], SCC binning by size, and subgraph marking of updated latch state pins.

To enable a more efficient data layout for GPU-based AT re-propagation on SCCs, Algorithm 3 bins the SCCs in each level into single-pin and multi-pin regions in parallel. The design is motivated by the observation that levelized SCCs vary largely in size, with single-pin ones dominating in practical circuits. Figure 5 shows an example of SCC binning.

The subgraph marking identifies the downstream SCCs of pins in the set P, limiting the scope of AT re-propagation to the affected subgraph. To implement this, Algorithm 4 uses a two-phase approach: it first marks SCCs with updated pins, then propagates marks level by level to their downstream neighbors. Figure 6 demonstrates how it marks dynamically determined subgraph rather than the entire circuit. The algorithm leverages SCC hierarchy, allowing each SCC to examine only its direct predecessors, and is well suited for GPU-level parallelism.

These steps collectively optimize the data layout and computation scope for later SCC-based steps. Together, they enhance GPU efficiency and ensure heterogeneous execution.

C. AT Re-Propagation on SCCs

This step re-propagates ATs on the downstream subgraph of pins in P, ensuring convergence to correct values. It proceeds level by level over marked SCCs using prior results from SCC levelization and subgraph marking. Due to size variations of SCCs, single-pin SCCs allow direct input arc relaxation, while multi-pin SCCs require iterative propagation. A naive thread-per-SCC strategy suffers from divergence and workload imbalance, making a tailored GPU approach necessary.

To address this, we propose a parallel strategy with mixed granularity that decouples the AT propagation for single- and multi-pin SCCs. Based on SCC binning, Algorithm 5 performs level-by-level propagation by concurrently launching two GPU kernels per level: one for single-pin SCCs (Algorithm 6) and one for multi-pin SCCs (Algorithm 7). The dual-stream model uses CUDA stream parallelism and synchronizes among levels.

Algorithm 5: AT-Re-Propagation-on-SCCs

Input : Arrival Times $\&AT_{max}$, Timing Graph G, SCC Levelization L, SCC Size Boundary SB Actual Time Borrows $\&ATB$

for *each* l *in* L **do** ▷ Re-propagate ATs level by level
  Launch AT_ReProp_Single_Pin_SCCs kernel on $cudaStream\_S$ with $(l, \&AT_{max}, G, SB, \&ATB)$
  Launch AT_ReProp_Multi_Pin_SCCs kernel on $cudaStream\_M$ with $(l, \&AT_{max}, G, SB, \&ATB)$
  synchronize $cudaStream\_S, cudaStream\_M$
  destroy $cudaStream\_S, cudaStream\_M$

Algorithm 6: AT-ReProp-on-Single-Pin-SCCs

Kernel Function AT_ReProp_Single_Pin_SCCs:
  $sccID \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
  if $sccID \geq SB[l]$ **then return**
  if $l[sccID]$ *is not Updating* **then return**
  $p \leftarrow l[sccID].get\_pins()$
  Relax all $p[0]$'s fanin arcs, update AT_{max} and ATB.

For single-pin SCCs, Algorithm 6 assigns **one GPU thread** per SCC to perform input arc relaxations. Each thread directly updates the AT of the single pin by relaxing its fanin arcs. This thread-per-SCC mapping aligns with the uniform workload of single-pin SCCs, achieving near-ideal thread utilization.

For multi-pin SCCs, Algorithm 7 assigns **one GPU block** per SCC to run iterative AT propagation until convergence. Threads in each block share and process the pin queue until it is empty. Each GPU thread is assigned to a pin in *queue* on demand. For each pending pin, the kernel 1) relaxes all unrelaxed input arcs from upstream SCCs, 2) then performs atomicRelax on intra-SCC fanout arcs to prevent data races during concurrent AT updates on one pin by multiple threads. Updated pins with changed ATs are atomically enqueued back into *queue* if they are not in it. The algorithm also detects divergence: following [2], propagation is guaranteed to diverge if any pin is enqueued more times than its SCC size.

This mixed-grained strategy combines thread-level agility for single-pin SCCs with block-level parallelism for multi-pin SCCs. It achieves better load balance and maximizes GPU utilization. As a result, synchronization overhead is reduced and timing analysis is accelerated for latch-intensive circuits.

Algorithm 7: AT-ReProp-on-Multi-Pin-SCCs

Kernel Function AT_ReProp_Multi_Pin_SCCs:
  $sccID \leftarrow blockIdx.x + SB[l]$
  if $sccID \geq$ size(l) **then return**
  if $l[sccID]$ *is not Updating* **then return**
  initialize $queue \leftarrow l[sccID].get\_pins()$
  while *queue is not empty* **do**
    $threads\_num \leftarrow \min(\text{size}(queue), blockDim.x)$
    if $threadIdx.x \geq threads\_num$ **then return**
    $p \leftarrow queue[threadIdx.x]$ and dequeue p
    for *each* p*'s unrelax fanin arc* $e(from, p)$ *in* G **do**
      if p *and* $from$ *are not in the same* SCC **then**
        Relax$(e, \&AT_{max}, \&ATB)$
    for *each* p*'s fanout arc* $e(p, to)$ *in* G **do**
      if p *and* to *are in the same* SCC **then**
        atomicRelax$(e, \&AT_{max}, \&ATB)$
      if $AT_{max}[to]$ *is unchanged* **then continue**
      if to *is Enqued* **then continue**
      Add$(enq_cnt[to], 1)$, atomicEnque$(queue, to)$
      if $enq\_cnt[to] >$ size$(l[sccID])$ **then**
        Report divergent loops.
    synchronize all threads in the block $blockIdx.x$

IV. Experimental Results

A. Experimental Setup

We developed our latch-aware STA framework, named HeteroLatch, designed as a heterogeneous timing analysis engine specifically for latch-based circuits. HeteroLatch leverages high-performance C++, CUDA, and Rust to ensure efficient computation, and it offers a flexible user interface.

To systematically evaluate the runtime benefits of HeteroLatch on latch timing analysis, we compared its performance against three representative baselines: (1) Synopsys PrimeTime (2021.06), a commercial industry standard; (2) the latest version of OpenSTA; and (3) a state-of-the-art SCC-based latch STA engine from [4]. GPU-based STA tools are excluded due to their lack of full latch support. Table II presents benchmark statistics of seven industrial latch-based designs. The first four circuits use 14nm technology, while the rest are 28nm. Among them, `case3` and `case4` natively incorporate latches with latch usage rates of around 30% and 40%, respectively; the remaining benchmarks, originally flip-flop-based, are modified to include 20% latch replacement. `case7` features multi-clock domains, unlike the single-clocked others.

Our experimental environment is a CentOS 7.9 Linux server with a 64-core Intel Xeon Platinum 8358 CPU and 8 NVIDIA A800 GPUs. The server is configured with 1 TB of system memory, and each GPU provides 80 GB of dedicated memory. Runtime is measured by the wall-clock time of the `report_timing` command after loading design files.

For fair comparison, all designs were evaluated using identical netlists, Liberty files, SDF delays, and SDC constraints. All experiments target setup paths; HeteroLatch also supports accurate hold checks, though they are not the focus of this work. All results in Table II were obtained under matched path slacks with PrimeTime, the golden sign-off standard, confirming HeteroLatch's full accuracy during acceleration.

B. Full Timing Performance

According to runtimes in Table II, HeteroLatch outperforms three baselines on all tested benchmarks. Speedup ratios range from 3.17–47.58× over PrimeTime, 5.96–16.19× over OpenSTA, and 1.60–2.29× over [4], with average gains of 12.64×, 9.45×, and 1.96×, respectively.

TABLE II: Runtime comparison between PrimeTime, OpenSTA, work in [4] and HeteroLatch on benchmarks.

Benchmark	Circuit Statistics			PrimeTime (16C)		OpenSTA (16C)		[4] (16C)		Ours GPU (16C + 1G)	
	#Gates	#Nets	#Pins	RT (s)	Ratio	RT (s)	Ratio	RT (s)	Ratio	RT (s)	Ratio
case1	770K	913K	3.04M	20.336	5.37	30.661	8.10	8.656	2.29	3.786	1.00
case2	737K	848K	2.92M	20.678	4.96	24.844	5.96	7.143	1.71	4.170	1.00
case3	131K	173K	482K	19.476	19.73	15.983	16.19	1.582	1.60	0.987	1.00
case4	891K	1.62M	3.14M	211.306	47.58	55.217	12.43	9.888	2.23	4.441	1.00
case5	2.57M	2.58M	8.96M	41.922	3.48	89.252	7.40	25.902	2.15	12.057	1.00
case6	4.30M	4.67M	16.8M	70.416	3.17	147.329	6.63	45.205	2.03	22.216	1.00
case7	3.47M	3.85M	13.7M	78.927	4.16	—	—	32.770	1.73	18.972	1.00
Avg. Ratio	—			—	12.64	—	9.45	—	1.96	—	1.00

RT: runtime in seconds. **Ratio**: runtime ratio compared to HeteroLatch on GPU.
All baselines are run with 16 CPU cores and 1 GPU, reporting k = 1000 paths. Note that due to inconsistencies in OpenSTA's path reporting rules across multiple clock domains, the last benchmark was not tested.

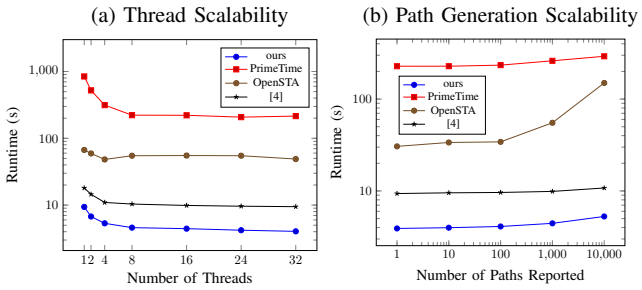

Fig. 7: Runtime scalability of four timers on design **case4** (a) under varying CPU threads, reporting k = 1000 paths, (b) with different numbers of paths k reported, using 16 CPU threads.

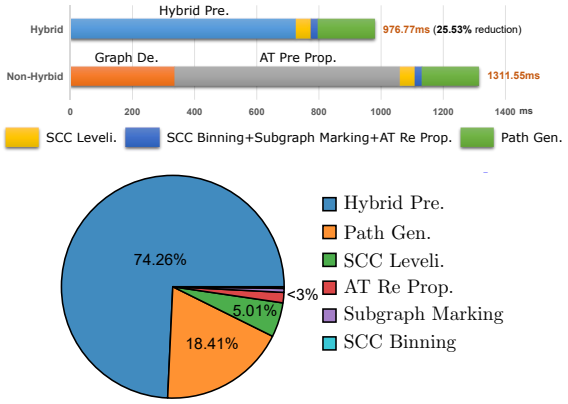

Fig. 8: Ablation study on design **case3** for CPU-GPU Preliminary Timing Analysis and the runtime breakdown.

C. Runtime Scalability Analysis

We further analyzed the runtime scalability of HeteroLatch and baseline methods under varying CPU thread counts (Figure 7(a)) and path numbers (Figure 7(b)).

Regarding CPU thread scalability, OpenSTA saturates near 4 threads, while PrimeTime scales rapidly from 1 to 8 threads but plateaus between 8 and 32, peaking at 24. Work [4] shows limited scalability, with 43% runtime reduction from 1 to 8 threads. In contrast, HeteroLatch achieves 53% reduction to 16 threads, reaching minimum at 32. Ultimately, all methods face limits due to inherent CPU parallelism constraints.

For top-k critical path generation, HeteroLatch maintains low runtime to k=10,000 with lower incremental cost than baselines. This efficient scaling mirrors prior GPU-accelerated timers [16, 18] using parallel prefix-suffix expansion.

D. Ablation Study and Runtime Breakdown

We investigated the performance of HeteroLatch via an ablation study on the part of hybrid CPU-GPU preliminary timing analysis (Section III A) and analyzing the runtime breakdown. As shown in Figure 8, we decompose the total analysis runtime for the design `case3` into its constituent steps. The results demonstrate that CPU-GPU parallel collaboration reduces the total runtime by 25.53% compared to the non-hybrid approach, with CPU execution time fully masked by overlapping GPU computations. Notably, the subgraph search, SCC binning, and AT re-propagation procedures account for a negligible fraction of the runtime (0.57%, 0.24%, and 1.51%, respectively). This minimal overhead confirms the runtime efficiency of our work.

V. Conclusion

This paper presents HeteroLatch, a CPU-GPU heterogeneous STA engine addressing latch-induced timing challenges, time borrowing and latch loops, while enabling GPU acceleration. By harmonizing adaptive SCC-based analysis and hierarchical timing propagation, our framework masks the overhead of graph decomposition through CPU-GPU task overlap and early termination, and balances GPU workloads. Experimental results show average speedups of 12.64× over PrimeTime, 9.45× over OpenSTA, and 1.96× over SOTA method [4], with full accuracy. HeteroLatch bridges the gap between latch-specific timing complexities and GPU-native parallelism. This work unlocks efficient verification for advanced-node circuits with dense latch clusters, supporting broader adoption of latch-centric methods for high-frequency, low-power designs.

REFERENCES

[1] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media, 2009.

[2] Synopsys, *PrimeTime, version: S-2021.06-SP1*, Synopsys, Inc., Mountain View, California, USA, 2021, synopsys, Inc. Software. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html

[3] "OpenSTA, version: 2.5.0," https://github.com/abk-openroad/OpenSTA.

[4] X. Shi, Z. Guo, Y. Lin, R. Wang, and R. Huang, "Handling latch loops in timing analysis with improved complexity and divergent loop detection," in *2025 IEEE/ACM Design, Automation and Test in Europe (DATE)*, 2025.

[5] S. Hassoun, C. Cromer, and E. Calvillo-Gámez, "Static timing analysis for level-clocked circuits in the presence of crosstalk," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 9, pp. 1270–1277, 2003.

[6] B. Li, N. Chen, Y. Xu, and U. Schlichtmann, "On timing model extraction and hierarchical statistical timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 3, pp. 367–380, 2013.

[7] X. Yuan and J. Wang, "Statistical timing verification for transparently latched circuits through structural graph traversal," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2010, pp. 663–668.

[8] B. Li, N. Chen, and U. Schlichtmann, "Statistical timing analysis for latch-controlled circuits with reduced iterations and graph transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 11, pp. 1670–1683, 2012.

[9] H. H.-W. Wang, L. Y.-Z. Lin, R. H.-M. Huang, and C. H.-P. Wen, "Casta: Cuda-accelerated static timing analysis for VLSI designs," in *Proc. ICPP*. IEEE, 2014, pp. 192–200.

[10] Z. Guo, T.-W. Huang, Z. Jin, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous static timing analysis with advanced delay calculator," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[11] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "Gcs-timer: Gpu-accelerated current source model based static timing analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[12] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. ASPDAC*. IEEE, 2009, pp. 260–265.

[13] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proceedings of the 39th international conference on computer-aided design*, 2020, pp. 1–9.

[14] ——, "Accelerating static timing analysis using cpu–gpu heterogeneous parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4973–4984, 2023.

[15] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W.-L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, "G-pasta: Gpu-accelerated partitioning algorithm for static timing analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[16] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 721–726.

[17] Z. Guo, T.-W. Huang, and Y. Lin, "Heterocppr: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[18] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated critical path generation with path constraints," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[19] G. Guo, T.-W. Huang, and M. Wong, "Fast sta graph partitioning framework for multi-gpu acceleration," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[20] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4219–4232, 2023.

[21] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, "Heteroexcept: A cpu-gpu heterogeneous algorithm to accelerate exception-aware static timing analysis," in *Proc. ICCAD*, 2024.

[22] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.

[23] R. Chen and H. Zhou, "Statistical timing verification for transparently latched circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1847–1855, 2006.

[24] K. Zhu, X. Di, W.-S. Luk, L. Wang, and J. Tao, "A fast timing analysis and optimization for latch-based circuits," in *2022 China Semiconductor Technology International Conference (CSTIC)*. IEEE, 2022, pp. 1–3.

[25] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[26] J. L. Bentley, *Programming Pearls*, 2nd ed. Addison-Wesley, 2000.