SSTA-X: GPU-Accelerated First-Order Block-Based Statistical Static Timing Analysis

Chih-Chun Chang, Yi-Hua Chung, Wan Luan Lee, Boyang Zhang, and Tsung-Wei Huang, *Member, IEEE*

***Abstract*—Statistical static timing analysis (SSTA) is a widely used approach to evaluate circuit timing by modeling gate and path delays as random variables affected by process variations. While Monte Carlo-based SSTA methods offer high accuracy, their reliance on massive sampling leads to very long runtime. To alleviate this runtime overhead, first-order block-based SSTA (FB-SSTA) has been proposed to approximate timing distributions through linear propagation of statistical moments. However, existing FB-SSTA solutions are largely limited to CPU-based parallelism, and their scalability can no longer quickly respond to the ever-growing size of SSTA problems. To overcome this limitation, we propose *SSTA-X*, a GPU-accelerated FB-SSTA algorithm that significantly enhances the runtime performance by leveraging the massive parallelism of modern GPU. SSTA-X introduces GPU-efficient data structures and kernel algorithms to accelerate key yet time-consuming operations in statistical delay, slew, and moment propagation. Experimental results show that SSTA-X achieves 3.10×–12.47× speedup over a CPU-based baseline on a large-scale design with millions of gates.**

***Index Terms*—Graphics processing unit (GPU) acceleration, statistical static timing analysis (SSTA).**

I. Introduction

Statistical static timing analysis (SSTA) enhances static timing analysis (STA) by modeling on-chip process variations (OCV) as random variables, enabling more accurate delay estimation [1, 2, 3]. Among various SSTA solutions, *Monte Carlo-based* SSTA (MC-SSTA) has been extensively studied. For instance, [4] proposes an MC-SSTA-based framework by leveraging quasi-Monte Carlo (QMC) sampling and Karhunen-Loéve expansion to reduce dimensionality; [5] introduces a stratified hybrid QMC sampling technique to further improve efficiency in incremental MC-SSTA. While these methods achieve high accuracy without making strong assumptions about delay distributions, their runtime remains a significant concern especially when analyzing large circuits, which require thousands to millions of samples to converge. To address this challenge, [6] and [7] implement MC-SSTA on GPU and FPGA platforms, respectively, achieving significant runtime improvement over CPU-based implementations.

While MC-SSTA offers high accuracy, its reliance on extensive sampling still results in very long runtime. To alleviate the long runtime, *first-order block-based* SSTA (FB-SSTA) has been proposed to approximate timing distributions through linear propagation of statistical moments [8]. Unlike MC-SSTA, FB-SSTA analyzes first-order sensitivities of all timing quantities to each variation source, eliminating the need for time-consuming sampling processes. Figure 1 shows an FB-SSTA example on a circuit of four gates. Timing quantities and variation sensitivities for interconnects and cells are stored in structured arrays. Different sensitivities quantify how much a timing quantity (e.g., delay) changes in response to variations, such as voltage fluctuations (ΔV), temperature shifts (ΔT), and channel length deviations (ΔL). During timing propagation through a combinational cell, each timing arc contributes its intrinsic delay and the associated interconnect delay to the arrival times and slews of its fan-in signals. The output arrival time and slew are then computed using statistical min/max operations under the tightness probability model [9, 10]. For sequential elements, such as flip-flops, the arrival times and required arrival times are further constrained by setup and hold conditions, which are modeled as functions of the input slews at the clock and data pins. Because all sum and max operations are performed in a linear canonical form, FB-SSTA is significantly more efficient than sampling-based methods.

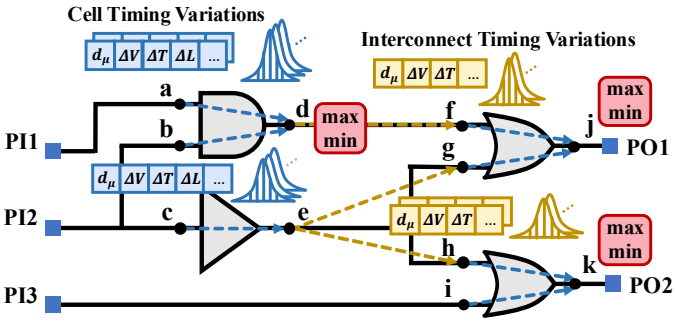


Fig. 1. Illustration of a first-order block-based SSTA (FB-SSTA). Sensitivities to variations for combinational cells (blue) and interconnects (yellow) are stored in arrays. The propagation of statistical quantities across circuit nodes and variation dimensions exhibits significant data parallelism that can benefit from GPU computing.

A number of works have been proposed to improve the accuracy and scalability of FB-SSTA. For example, the UCLA Timer [11] and the extended pseudo-canonical model [12] enhance first-order representations by incorporating spatial correlations and hierarchical variation models. The skew-normal canonical delay model [13] and conditional linear min/max approximation [14] further improve max operation accuracy while maintaining tractable runtime. IITiMER [15] implements a CPU-parallel algorithm to accelerate the propagation of statistical quantities. Despite these improvements, existing FB-SSTA solutions are largely limited to CPU parallelism, and their scalability typically saturates at 8–16 threads, making them unsuitable for large FB-SSTA problems.

Observing from Figure 1, the propagation pattern in FB-

SSTA exhibits substantial data parallelism, enabled by the concurrent computation of statistical quantities across multiple circuit components and variation dimensions. Compared to CPUs, modern GPUs support orders of magnitude more parallelism and much higher memory bandwidth. The large amount of data parallelism exhibited by FB-SSTA provides a unique opportunity to improve the analysis runtime by leveraging the power of GPU. Nevertheless, designing a GPU-parallel FB-SSTA is very challenging. First, the irregular structure of timing graphs introduces non-coalesced and scattered memory accesses, which can severely degrade GPU performance. Second, although the dimensionality of variation modeling is moderate, achieving high GPU occupancy while maintaining low memory latency requires careful data layout and memory access optimization. Last but not least, statistical operations such as min and max over distributions are inherently nonlinear and involve data-dependent branching, which complicates parallel execution and limits warp efficiency.

To overcome these challenges, we propose *SSTA-X*, a GPU-accelerated FB-SSTA algorithm that employs GPU-efficient data structures and optimized kernel algorithms to significantly accelerate FB-SSTA runtime. We summarize three technical contributions below:

- **Topology-aware memory layout**: We design a GPU-efficient data layout aligned with the topological structure of the timing graph. This layout improves spatial locality and enables coalesced memory access on GPU, effectively mitigating the performance penalties of irregular access patterns.
- **Latency-optimized data access**: We design a GPU-efficient kernel for computing timing quantities that leverages vectorized memory accesses to retrieve variation data, thereby reducing memory latency and improving overall bandwidth utilization.
- **Warp-efficient statistical operations**: We design warp-optimized GPU kernels for timing quantity propagation and statistical operations, enabling fast intra-warp communication and reducing memory traffic through efficient data sharing among threads.

We evaluate the performance of SSTA-X on a set of industrial SSTA benchmarks from the TAU Contests [16, 17]. Compared to the first-place winner, IITiMER [15], SSTA-X achieves a 3.10×–12.47× speedup on large-scale designs with millions of gates and nets. To the best of our knowledge, no open-source FB-SSTA tool currently supports GPU acceleration. We plan to release SSTA-X as open-source to benefit the timing analysis research community.

II. Statistical Static Timing Analysis

SSTA models a circuit as a *directed acyclic graph* (DAG), where nodes represent logic pins and edges represent timing arcs. Unlike STA, which assumes deterministic delays, SSTA treats delays as random variables and propagates them through the DAG using statistical operations. Mainstream SSTA techniques fall into two categories: *sampling-based* and *parametric*. In sampling-based SSTA [1], timing quantities are computed via stochastic techniques such as Monte Carlo sampling, where each sample runs deterministic STA with specific parameters. While this method accurately captures non-linear effects like statistical min and max, it incurs high computational cost when thousands of samples are needed for large circuits.

Parametric SSTA, on the other hand, analytically models each timing quantity, such as delay, arrival time, and slack, as linear combinations of independent and correlated Gaussian variables that capture process and environmental variations (e.g., voltage, temperature, channel length, and metal width). This first-order formulation can be expressed as:

$$T = a_0 + \sum_{i=1}^{n} a_i \Delta X_i + a_{n+1} \Delta R, \tag{1}$$

where a_0 denotes the nominal value, ΔX_i represents zero-mean global sources of variation, ΔR corresponds to independent local randomness, and each coefficient a_i quantifies the sensitivity to the respective variation source. This linear parametric form is propagated analytically through the timing graph to evaluate the sensitivity of timing paths to individual variation sources, facilitating robust design optimization. Among parametric approaches, block-based SSTA propagates canonical delay models across the entire circuit by performing a levelized, breadth-first traversal of the timing graph [8], in contrast to path-based methods that analyze only a selected subset of timing paths. A critical challenge in block-based SSTA is accurately computing statistical operations, especially the min/max operation at fan-in convergence points, since the maximum of two Gaussian random variables does not yield a Gaussian distribution. To address this, Clark's method [10] is widely adopted, approximating the statistical maximum by matching the first two moments of the resultant distribution. Specifically, consider Gaussian-distributed random variables $X \sim N(\mu_X, \sigma_X^2)$ and $Y \sim N(\mu_Y, \sigma_Y^2)$, where μ_X, μ_Y denote their means, σ_X, σ_Y denote their standard deviations, and ρ represents their correlation coefficient. The operation $Z = \max(X, Y)$ produces a non-Gaussian distribution, whose mean μ_Z and standard deviation σ_Z can be analytically approximated using Clark's method. First-order block-based (FB-SSTA) has proven particularly effective for large designs and early-stage optimization, where fast yet reasonably accurate feedback is essential.

A. Parallel SSTA

Despite its advantages in accuracy, SSTA often incurs significantly higher runtime than STA due to the overhead of statistical operations. For instance, enabling SSTA in an industry-standard timer can degrade runtime by at least 5×–10× even for relatively small designs [16]. To address this runtime challenge, several works have explored parallelizing SSTA: Gulati [6] implemented MC-SSTA on GPUs, achieving up to 260× speedup over serial CPU versions; Veetil [18] proposed a GPU-accelerated SH-QMC approach with 48× speedup over quad-core CPUs; Cong [7] demonstrated FPGA-based acceleration for MC-SSTA, reporting up to 100× improvement. While these efforts highlight the potential of

hardware acceleration, they primarily focus on sampling-based techniques.

In contrast, most prior work on parametric SSTA has focused on improving timing accuracy by introducing more refined statistical delay models [19, 20]. More recently, IITiMER [15] addressed the runtime bottleneck by leveraging multi-threading to parallelize FB-SSTA. However, its scalability remains limited to CPU-based parallelism, with performance gains typically saturating at 8–16 threads. As shown in Figure 2, IITiMER's runtime plateaus at 16 threads on large circuits due to limited thread resources, constrained memory bandwidth, and high threading overhead, making it difficult for CPUs to exploit the substantial data parallelism inherent in SSTA. Meanwhile, recent work such as INSTA [21] demonstrates GPU-accelerated deterministic STA with OCV/CPPR support, reflecting growing interest in GPU-based timing engines. However, INSTA targets deterministic STA rather than parametric SSTA and therefore addresses a different problem space.

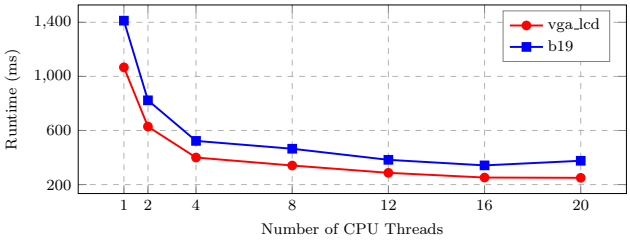


Fig. 2. Scalability of CPU-parallel IITiMER on two circuits (vga_lcd and b19), which saturates at 8–16 CPU threads.

To better understand the limitations of CPU-parallel SSTA, we profile the runtime breakdown of IITiMER in Figure 3. The analysis shows that timing propagation dominates execution time, accounting for more than 50% of the total runtime. This stage computes not only nominal arrival and required times, but also first-order sensitivities to multiple variation sources across early/late modes and rise/fall transitions, effectively multiplying the computational workload by four. The complexity further increases with the number of variation parameters, as each timing quantity is represented in a vectorized canonical form containing one coefficient per source. Despite this apparent overhead, these computations are structurally independent across gates, transitions, timing modes, and variation dimensions, offering substantial fine-grained parallelism. Such inherent concurrency makes timing propagation particularly well-suited for high-throughput acceleration on modern GPUs.

To explore this potential, we adopt the widely recognized FB-SSTA formulation originally introduced in the TAU Contest [16], as it provides a rigorous, open framework for evaluating both algorithmic correctness and computational efficiency. Importantly, our focus on this formulation does not limit generalizability. FB-SSTA remains the foundation of many modern SSTA tools due to its analytical tractability and first-order sensitivity propagation model. Furthermore, most block-based SSTA algorithms share the same core structure, including levelized graph traversal and repeated `max`/`sum` operations over canonical delay models. Therefore, the techniques and performance insights presented in this work are broadly applicable to a wide range of current and future SSTA engines that employ similar propagation mechanisms.

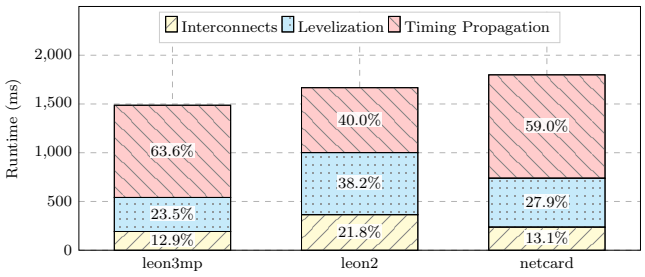


Fig. 3. Runtime breakdown of IITiMer on three large circuits, leon3mp, leon2, and netcard. The total runtime is divided into three phases: circuit levelization, interconnect timing computation, and timing propagation (including forward and backward).

B. Challenges of GPU Acceleration

We highlight three major challenges in designing a GPU-accelerated FB-SSTA algorithm:

- **Irregular computation patterns**: The irregular structure of timing graphs, which are modeled as DAGs, introduces significant challenges for parallelization. Since SSTA must respect the dependency constraints of the circuit graph, different data become ready for computation at different times. This results in workload imbalance and warp divergence, both of which degrade the efficiency of Single Instruction, Multiple Threads (SIMT) execution on GPU.
- **Scattered memory access patterns**: FB-SSTA requires frequent memory accesses to propagate arrival times and sensitivities across multiple operating modes (early/late) and signal transitions (rise/fall). Due to the irregular graph topology, these accesses are often scattered and poorly aligned, which can lead to poor memory coalescing and thus increase memory traffic between global memory and L1/L2 caches.
- **Control-flow divergence**: Statistical max operations are inherently non-linear and depend on distribution parameters such as mean and variance. These operations often involve data-dependent control flow, causing threads within a warp to follow divergent execution paths when evaluating distinct timing scenarios. This divergence degrades the efficiency of parallel execution and presents a fundamental challenge in implementing high-throughput statistical operations on GPU.

These challenges necessitate careful kernel design, memory layout optimization, and warp-efficient implementation to fully leverage GPU parallelism for FB-SSTA.

III. Algorithm

To address the above challenges, we introduce SSTA-X, a GPU-accelerated algorithm for fast and accurate FB-SSTA. SSTA-X introduces three key innovations: a topology-aware memory layout, latency-optimized data access, and warp-efficient statistical operations to accelerate FB-SSTA on GPU.

The memory layout of SSTA-X is carefully crafted to mirror the topological structure of the timing graph. This alignment improves spatial locality and enables coalesced global memory access, effectively mitigating the performance degradation caused by irregular memory access patterns typically encountered in SSTA. To further reduce memory latency, SSTA-X utilizes vectorized memory loads within the timing computation kernels, allowing concurrent retrieval of multiple variation-aware parameters and maximizing global memory bandwidth usage. Furthermore, SSTA-X incorporates warp-optimized GPU kernels that leverage intra-warp communication primitives to eliminate redundant memory accesses and support efficient data sharing during the statistical operations in both forward and backward propagation phases. Based on our industrial experience and collaboration with major EDA vendors, we observe that these GPU kernels are compatible with the analytic formulations used in commercial SSTA engines, which typically employ variants of Clark's method with additional sign-off considerations such as vendor-specific correlation models. Consequently, SSTA-X can be extended to commercial timing methodologies without requiring substantial changes to the underlying GPU parallelization strategy.

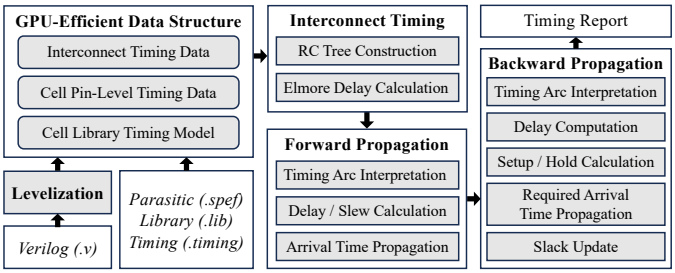
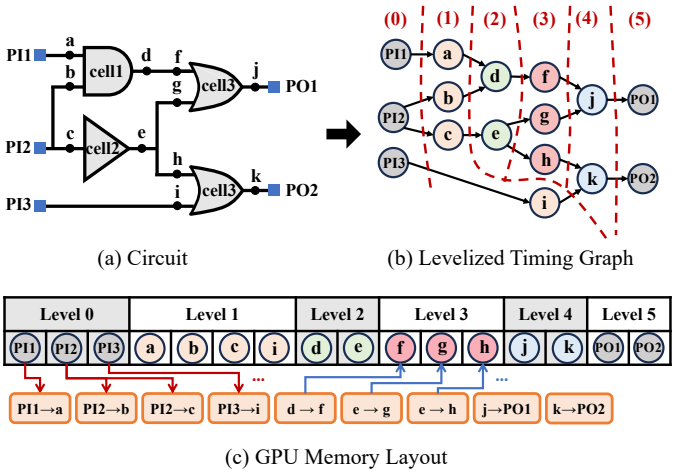


Fig. 4. Overview of *SSTA-X*

Figure 4 presents an overview of SSTA-X. The framework constructs the timing graph, builds RC trees for interconnects, and then performs forward and backward propagation using the GPU kernels described in Sections III-B – III-D. The internal data representation preserves the topological order of circuit elements and separates interconnect timing, cell timing, and library models to enable efficient parallel traversal on GPU.

A. GPU-efficient Data Structure

To enable high-throughput execution on GPUs, we first transform the hierarchical circuit representation into a flattened, GPU-efficient memory layout, ensuring predictable and highly parallel access patterns. This begins with a parallel breadth-first traversal of the timing graph to assign each instance a topological level. As illustrated in Figure 5, all instances within the same level are independent and can therefore be processed concurrently. Timing graphs may contain nodes with multiple fan-in arcs driven by different primary inputs or intermediate nodes. Our levelization scheme naturally handles such multi-source dependencies by assigning each node the maximum level among its fan-ins, ensuring that all predecessors have been processed before the node itself.



(a) Circuit (b) Levelized Timing Graph

(c) GPU Memory Layout

Fig. 5. (a) Example circuit with primary inputs (PIs) and outputs (POs). (b) Corresponding levelized timing graph, where nodes are grouped by their distance from the inputs for parallel processing. (c) GPU memory layout reordered according to the levelized structure to enhance spatial locality and enable coalesced memory access.

This avoids ambiguity in level assignment, preserves correct synchronization, and prepares the structure for uniform GPU traversal.

After levelization, timing data for each instance, pin, and interconnect is linearized into contiguous arrays to remove irregular pointer chasing and enable coalesced memory access. The preprocessing step flattens hierarchical library information such as pin-to-pin arcs, setup/hold constraints, and interconnect parasitics into three compact tables, each supported by pointer arrays recording per-pin or per-arc offsets. These pointer structures preserve logical hierarchy while enabling deterministic, offset-based indexing essential for efficient GPU kernels. The resulting tables include (1) combinational-cell tables storing per-arc delay and slew data, (2) sequential-cell tables capturing flip-flop and latch constraints, and (3) interconnect tables encoding RC node ordering, fan-in/fan-out structure, and per-tap delay/slew. During preprocessing, we also construct compact lookup maps linking the original netlist hierarchy to the flattened layout (e.g., mapping gate names, pins, and interconnect segments to array indices). All per-level arrays are aligned to 32-element boundaries to maximize coalesced access, and timing quantities (arrival time, required time, and slew) are vectorized using `float4` to load nominal and sensitivity coefficients in a single instruction. These transformations convert irregular netlist structures into deterministic, level-wise parallel layouts with minimal synchronization overhead, forming the foundation for the high-performance GPU kernels described in later sections.

Following the TAU Contest specification, arrival time and slew at each instance are represented across four timing modes, early/late and rise/fall, as illustrated in Figure 7. Each quantity is modeled as a first-order parametric expression:

$$t = \mu + \sum_{i=1}^{6} \alpha_i \Delta_i + \alpha^{\Delta R} \Delta R \tag{2}$$

where μ is the nominal timing value and $\Delta_i \sim \mathcal{N}(0,1)$

represent seven variation sources: voltage, temperature, channel length, device width, threshold voltage, metal variation, and intra-die random variation. As shown in Figure 7, each pin stores 32 floating-point values (one nominal and seven sensitivities for each of the four modes).

For interconnect modeling, we use CUDA `float4` types to store rise and fall delay/slew values and their sensitivities, following the TAU convention that metal variation is the only interconnect-related variation source. This vectorized layout enables each GPU thread to load or update multiple timing quantities in a single instruction, improving effective bandwidth via coalesced memory access and aligning data to warp-sized (32 element) blocks.

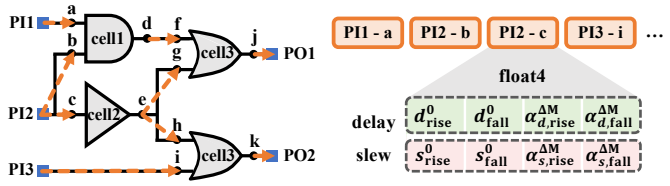


Fig. 6. The interconnect data structure stores rise/fall delay and slew using CUDA `float4` types to represent both nominal values and variation sensitivities.

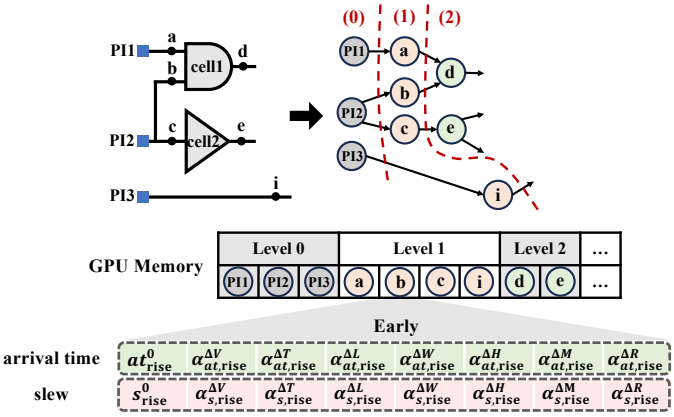


Fig. 7. The data structure stores arrival time and slew across early/late modes and rise/fall transitions for each pin. Each timing quantity includes one nominal value and seven variation sensitivities, totaling 32 values per pin.

Although SSTA-X adopts the TAU variation model, our framework is not restricted to this specification. All delay, slew, and statistical kernels operate on vectorized sensitivity coefficients, making it straightforward to incorporate additional variation sources required in advanced nodes such as FinFET-induced fin-geometry fluctuations, workfunction metal granularity technologies, or residual random dopant effects, simply by expanding the dimensionality of the sensitivity vectors.

B. Kernel Algorithms for Interconnect Timing Calculation

We approximate the interconnect delay and slew based on the Elmore delay model [16, 22, 23]. The nominal delay d^0 at node t is given by

$$d_t^0 = \sum_{k \in \text{path}} R_k \left(\sum_{j \in \text{downstream}(k)} C_j \right) \tag{3}$$

To incorporate the impact of metal layer variation on interconnect timing, we adopt a first-order sensitivity-based model that modifies resistance and capacitance values using library-provided scaling factors. Specifically, each metal layer's variation is modeled by a Gaussian variable ΔM, and its effect is captured through corner scaling factors m_R^σ and m_C^σ, which scale the nominal resistance and capacitance, respectively. These values are provided by the library and applied only to interconnect structures, as specified by the TAU contest. As specified, both the delay (d_t) and slew (s_t) of an interconnect segment affected by metal variation are expressed using linear sensitivity formulations:

$$d_t^{\Delta M} = d_t^0 + \alpha_d^{\Delta M} \Delta M \tag{4}$$

$$s_t^{\Delta M} = s_t^0 + \alpha_s^{\Delta M} \Delta M \tag{5}$$

Here, d_t^0 and s_t^0 are the nominal delay and slew values. $\alpha_d^{\Delta M}$ and $\%\alpha_s^{\Delta M}$ are first-order sensitivities that quantify how much the delay and slew vary per unit change in metal variation ΔM. These sensitivities are derived using finite differencing over a single standard deviation σ:

$$\alpha_d^{\Delta M} = \frac{d_t^{\Delta M} - d_t^0}{\sigma} \tag{6}$$

$$\alpha_s^{\Delta M} = \frac{s_t^\sigma - s_t^0}{\sigma} \tag{7}$$

The nominal slew s_t^0 is approximated using a moment-based approach derived from the Elmore delay model. Specifically, the slew is estimated from the second central moment β of the RC tree's impulse response:

$$s_t^0 \approx \sqrt{2\beta - d_t^{0\,2}} \tag{8}$$

Here, d_t^0 is the nominal Elmore delay, and β represents the second moment of the RC tree, computed as:

$$\beta = \sum_{k \in \text{path}} R_k C_k d_k \tag{9}$$

In this formulation, R_k and C_k are the resistance and capacitance of each RC segment, and d_k is the delay from the driver to node k. To compute the Elmore delay, traditional CPU implementations [15] typically perform a depth-first search (DFS) traversal of the RC tree to obtain both upstream and downstream paths. While DFS provides a natural way to walk the RC structure on CPUs, its recursive and irregular memory-access patterns make it poorly suited for GPU execution and prone to load imbalance when RC trees are deep or highly skewed. To address these limitations, SSTA-X replaces DFS with a flattened, BFS-based representation [24] that assigns each net to a cooperating GPU thread block and stores its RC tree as a linear parent-list. This structure guarantees

that every parent appears before its children, enabling both downstream and upstream dynamic-programming sweeps to run in a strictly ordered, coalesced fashion. As a result, RC computation avoids the bottlenecks inherent in CPU-based DFS implementations and ensures balanced execution even under diverse interconnect topologies.

Algorithm 1: RC Timing Kernel for Delay

```
   Input: N, M, K, Mc, Mr
   Input: net_st, tap_st, pres, pid, order, cap, tcap
   Output: load4, beta4, delay4
1  net_id ← thread's global ID
2  parallel for each thread
3      mc4 ← float4(1, 1, Mc, Mc)
4      mr4 ← float4(1, 1, Mr, Mr)
       // Initialization
5      for i ← net_st[net_id] to net_st[net_id+1] do
6          cap4 ← float4(cap[i], cap[i], cap[i], cap[i])
7          load4[i] ← cap4 × mc4
8          beta4[i] ← cap4 × mc4
9      end
       // Capacitances accumulation
10     for i ← tap_st[netId] to tap_st[netId+1] do
11         tcap4 ← float4(tcap[2i], tcap[2i+1], tcap[2i], tcap[2i+1])
12         load4[i] ← load4[i] + tcap4
13         beta4[i] ← beta4[i] + tcap4
14     end
       // Recursive load aggregation
15     for i ← net_st[net_id+1] − 1 down to net_st[net_id] do
16         n ← order[i] + net_st[net_id]
17         par ← pid[n] + net_st[net_id]
18         load4[par] ← load4[par] + load4[n]
19     end
       // Upstream delay calculation
20     for i = net_st[net_id] + 1 to net_st[net_id+1] do
21         n ← order[i] + net_st[net_id]
22         par ← pid[n] + net_st[net_id]
23         pres4 ← float4(pres[n], pres[n], pres[n], pres[n])
24         delay4[n] ← delay4[par] + mr4 × pres4 × load4[n]
25     end
26 end
```

Algorithm 1 presents the GPU kernel used in SSTA-X for computing Elmore delay and related timing quantities over flattened RC trees. Given the levelized structure and precomputed parent lists, the kernel operates independently on each net in parallel. The kernel takes as input: the number of nets, nodes, and taps (N, M, K); net and tap offset arrays (`net_st`, `tap_st`) to identify the range of each net and its associated taps; node-level resistances `pres`, capacitances `caps`, parent indices `pid`, and the topological order `order`; tap capacitances for rise/fall transitions `tcaps`; and metal variation scale factors M_c and M_r. The outputs consist of three `float4` arrays of size M: `load4`, `beta4`, and `delay4`, which store the downstream capacitance load, impulse response, and Elmore delay, respectively, for both rise and fall transitions under nominal and variation-aware conditions. Each `float4` vector represents `make_float4`(a, b, c, d), which packs four floating-point values for efficient vectorized operations. Lines 5–9 compute the initial capacitance load and impulse values for each node using broadcast `float4` operations to capture rise/fall values under nominal and metal variation conditions. To ensure correct indexing within each net, the node and parent indices are converted from relative to absolute positions using the offset `net_st[net_id]`, as indicated in the comment on Line 15. Lines 10–14 accumulate tap capacitance, as described in Equation (3). Lines 15–19 perform recursive load aggregation from children to parents following the topological order. Finally, lines 20–25 compute the upstream delay for each node by traversing from parent to child, applying scaled resistance and load values using vectorized operations.

Algorithm 2: RC Timing Kernel for Slew

```
   Input: Same inputs as in Algorithm 1
   Output: slew4
1  net_id ← thread's global ID
2  parallel for each thread
       // Corresponding delay multiplication
3      for i = net_st[net_id] to net_st[net_id+1] do
4          beta4[i] ← beta4[i] × delay4[i]
5      end
       // Downstream beta Aggregation
6      for i = net_st[net_id+1] − 1 down to net_st[net_id] do
7          n ← orders[i] + net_st[net_id]
8          par ← pid[n] + net_st[net_id]
9          beta4[par] ← beta4[par] + beta4[n]
10     end
       // Upstream slew calculation
11     for i = net_st[net_id] + 1 to net_st[net_id+1] do
12         n ← order[i] + net_st[net_id]
13         par ← pid[n] + net_st[net_id]
14         pres4 ← float4(pres[n], pres[n], pres[n], pres[n])
15         slew4[n] ← slew4[par] + mr4 × pres4 × beta4[n]
16     end
17 end
```

Algorithm 2 continues the parallel RC timing kernel by computing the slew values for each node. Given the previously computed delay and impulse values from Algorithm 1, the kernel first updates the beta term at each node by multiplying it with its corresponding delay (lines 3–5). The downstream beta accumulation, performed in lines 6–10, recursively aggregates the impulse from children to parent nodes in reverse topological order. Finally, lines 11–16 compute the upstream slew by propagating weighted beta values along the RC tree using scaled resistance. This propagation follows the topological order and enables efficient per-warp computation using vectorized operations, preserving the spatial structure established in the delay kernel.

Following the completion of delay and slew propagation, the sensitivities $\alpha_d^{\Delta M}$ and $\alpha_s^{\Delta M}$ for each tap node can be efficiently computed using finite differencing. As shown in Equations (6) and (7), these values are obtained by subtracting the nominal timing quantities from their perturbed counterparts under metal variation. The overall structure of the RC timing kernels enables high throughput and memory coalescing, while avoiding recursion and maintaining data locality on GPU.

Because this kernel operates on a flattened RC-tree representation, its correctness and efficiency depend on the parasitic topology of each net remaining fixed during an analysis run. When design modifications occur, only the RC trees of affected nets need to be reconstructed. The flattening procedure is linear in the size of the RC tree, allowing these updates to be applied selectively without rebuilding the entire design. This

property makes the kernel structure naturally compatible with future support for incremental ECO timing.

C. Kernel Algorithm for Cell Timing Calculation

To model variation-aware delay and slew for combinational cells, we adopt the linear parametric model described in the TAU Timing Contest. In this model, the delay D and output slew S_o of a cell are represented as linear functions of the input slew S_i, output load C_L, and sensitivities to process and environmental variations. The general form expresses as follows:

$$D = a(1 + \sum_{i=1}^{6} k_{d,i}\Delta_i + k_{d,r}\Delta R) + bC_L + cS_i \tag{10}$$

$$S_o = x(1 + \sum_{i=1}^{6} k_{s,i}\Delta_i + k_{s,r}\Delta R) + yC_L + zS_i \tag{11}$$

In these equations, a, b, c, and x, y, z denote the nominal coefficients that model the baseline behavior of delay and slew with respect to load and input slew. These coefficients are extracted from cell characterization data in the standard cell library. The terms $k_{d,i}$ and $k_{s,i}$ represent the first-order sensitivity coefficients with respect to global variation sources Δ_i, where $\Delta_i \sim \mathcal{N}(0, 1)$ for $i = 1, \ldots, 6$. These include supply voltage (ΔV), temperature (ΔT), channel length (ΔL), device width (ΔW), voltage threshold (ΔH), and metal interconnect (ΔM). The term ΔR captures random intra-die variation, with corresponding sensitivities $k_{d,r}$ and $k_{s,r}$. Since each variation term is multiplied by its respective coefficient, we can interpret $k_{d,i}$ or $k_{s,i}$ as the fractional impact of variation source Δ_i on the nominal value a or x, respectively. The total sensitivity of delay to a specific variation source (e.g., ΔV) must also include the effect of variation on the input slew S_i. This leads to the following augmented form:

$$\alpha_d^{\Delta V} = a \cdot k_{d,v} + c \cdot s_v \tag{12}$$

Here, s_v is the sensitivity of the input slew to the variation source ΔV, and c represents how input slew affects delay. This coupling ensures accurate modeling of variation propagation. To compute the delay sensitivity to uncorrelated random variation ΔR, we aggregate independent sources using a root-sum-square (RSS) approach:

$$\alpha_d^{\Delta R} = \sqrt{(a \cdot k_{d,r})^2 + (c \cdot s_r)^2} \tag{13}$$

This expression combines both direct and indirect contributions of random variation through intrinsic delay (via a) and via its influence on input slew (via c). This provides a complete characterization of cell-level variation effects in a form suitable for propagation.

Based on the cell model, we design a warp-parallel GPU kernel (Algorithm 3) in SSTA-X to efficiently compute variation-aware timing quantities for combinational cells. Each warp is assigned to one cell instance, and its 32 threads are partitioned into four 8-thread groups corresponding to the four timing arc types: early-rise, early-fall, late-rise, and late-fall. The kernel takes as input the number of timing arcs per level (N), the parametric coefficients for delay and slew, the load capacitance of each arc, and the input slew values, and outputs the resulting delay or slew values. According to each thread's *laneID*, the kernel loads the corresponding input slew (line 4) and sensitivity coefficients (lines 5–14), evaluates the fractional term in Equation (12) (lines 15–23), and applies the random-component adjustment in Equation (13) when needed (line 24). This organization enables efficient warp-level evaluation of gate delay and slew sensitivities during timing propagation.

By leveraging warp-level parallelism, the kernel assigns one thread to each scalar component of the parametric expression, allowing all nominal and sensitivity terms to be computed simultaneously within a single warp. This design enables full utilization of GPU SIMD resources and eliminates the need for shared memory or inter-warp communication. The implementation is highly efficient due to its memory access pattern and coalesced global memory loads. Each thread independently processes a scalar output, while all threads share per-arc data

Algorithm 3: Combinational Cell Kernel

Input: *N*, *dep*, *cap*, *slew*
Input: *dep_st, cap_st, slew_st, out_st*
Output: *out*
// *laneId*: thread index in a GPU warp
1 **parallel for each thread in a GPU warp**
// Arc type for early/late and rise/fall
2 $type \leftarrow \left\lfloor \frac{laneId}{8} \right\rfloor$
3 $typeId \leftarrow laneId \bmod 8$
4 $sv \leftarrow slew[slew\_ptr+laneId]$
// Load sensitivities for rise/fall
5 **if** $laneId < 16$ **then**
6 $dep0 \leftarrow dep[dep\_st]$
7 $dep1 \leftarrow dep[dep\_st+1]$
8 $dep2 \leftarrow dep[dep\_st+2]$
9 **end**
10 **else**
11 $dep0 \leftarrow dep[dep\_st+9]$
12 $dep1 \leftarrow dep[dep\_st+10]$
13 $dep2 \leftarrow dep[dep\_st+11]$
14 **end**
// Compute nominal delay/slew value
15 **if** $typeId == 0$ **then**
16 $out[out\_st + laneId] \leftarrow dep0 + dep1 \times cap[cap\_st + 2 \times (\frac{type}{2})] + dep2 \times sv$
17 **end**
// Compute sensitivities for canonical form
18 **if** $1 \le typeId \le 5$ **then**
19 $out[out\_st + laneId] \leftarrow dep0 \times dep[dep\_st + 9 \times (\frac{type}{2}) + 2 + typeId] + dep2 \times sv$
20 **end**
21 **if** $typeId == 6$ **then**
22 $out[out\_st + laneId] \leftarrow dep1 \times cap[cap\_st + 2 \times (\frac{type}{2}) + 1]$
23 **end**
// Compute the sensitivity for ΔR
24 **if** $typeId == 7$ **then**
25 $temp1 \leftarrow dep0 \times dep[dep\_st + 9 \times (\frac{type}{2}) + 8]$
26 $temp2 \leftarrow dep2 \times sv$
27 $out[out\_st+laneId] \leftarrow \sqrt{(temp1)^2 + (temp2)^2}$
28 **end**
29 **end**

such as input slew, output load, and sensitivity coefficients. Moreover, the kernel is readily extensible. If the number of sensitivity coefficients exceeds the current allocation (e.g., more than seven), the per-arc thread group can be expanded to accommodate additional terms. Although increasing beyond 8 threads per arc may misalign with warp boundaries, this can be addressed by either assigning multiple warps per arc or evaluating the sensitivities in multiple rounds.

To model the behavior of sequential elements, we adopt the flip-flop model defined in the TAU timing Contest. Similar to combinational cells, flip-flop setup and hold times are modeled as parametric linear functions of the input slews at the clock and data pins:

$$\begin{aligned} t_{\text{setup}} &= g + h \cdot S_{\text{CK}} + j \cdot S_D \\ t_{\text{hold}} &= m + n \cdot S_{\text{CK}} + p \cdot S_D \end{aligned} \tag{14}$$

where S_{CK} and S_D denote the input slew rates at the clock and data pins, and g, h, j, m, n, p are library-provided coefficients. Since these expressions follow the same linear parametric form as the cell delay and slew models, SSTA-X reuses the same GPU kernel used for combinational cells to evaluate flip-flop timing quantities. This unification simplifies the scheduling and computation logic while maintaining warp-level efficiency.

Although our experiments use the TAU linear parametric cell model, the SSTA-X framework is not restricted to this representation. In advanced-node design flows, cell delay and slew are often characterized using nonlinear delay model (NLDM) lookup tables. NLDM improves accuracy in nonideal operating regions (e.g., large fanout or degraded slews) but requires table lookup, bilinear interpolation, and sensitivity extraction—operations that increase per-arc arithmetic intensity. These computations map naturally to GPU. For example, table entries can be accessed with coalesced reads, interpolation is performed by parallel threads, and the added arithmetic intensity helps amortize memory latency. As a result, incorporating NLDM is expected to further widen the performance advantage of SSTA-X over CPU baselines. Because the GPU kernels in SSTA-X operate on vectorized timing coefficients, NLDM support primarily affects the coefficient-extraction stage, while the core GPU parallelization strategy remains unchanged.

D. Kernel Algorithm for Statistical Max Operation

In SSTA-X, we adopt a statistical max operation based on the tightness probability model [9, 10], which enables efficient and accurate propagation of parametric timing quantities under variations. Each timing quantity (e.g., arrival time or slew) is modeled as a Gaussian random variable with one nominal value and seven first-order sensitivity coefficients, corresponding to the parameters: ΔV, ΔT, ΔL, ΔW, ΔH, ΔM, and ΔR. To compute the statistical maximum between two such distributions, we first estimate the probability that one arrival time exceeds the other. Given two such arrival time distributions, $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ and $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$, the probability that X exceeds Y, known as the tightness probability T_X, is computed as:

$$T_X = \Phi\left(\frac{\mu_X - \mu_Y}{\theta}\right) \tag{15}$$

where

$$\theta = \sqrt{\sigma_X^2 + \sigma_Y^2 - 2\rho\sigma_X\sigma_Y} \tag{16}$$

and ρ is the correlation coefficient between X and Y. Using T_X, the expected value and variance of the statistical max $\max(X, Y)$ are computed as:

$$\mathrm{E}[\max(X,Y)] = \mu_X T_X + \mu_Y(1 - T_X) + \theta\,\phi\left(\frac{\mu_X - \mu_Y}{\theta}\right)$$

$$\begin{aligned} \mathrm{Var}[\max(X,Y)] = {} & (\sigma_X^2 + \mu_X^2)T_X + (\sigma_Y^2 + \mu_Y^2)(1 - T_X) \\ & + (\mu_X + \mu_Y)\theta\,\phi\left(\frac{\mu_X - \mu_Y}{\theta}\right) \\ & - \left(\mathrm{E}[\max(X,Y)]\right)^2 \end{aligned} \tag{17}$$

where $\phi(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$ is the standard normal probability density function, and $\Phi(x)$ is its cumulative distribution function (CDF).

If the variances of X and Y are perfectly correlated or have identical variances ($\theta = 0$), the statistical max simplifies to a conventional maximum between the two means. Otherwise, the computation follows the statistical blending formula. Specifically, the output sensitivities are computed as a weighted linear combination of the input sensitivities from X and Y, using the tightness probability T_X (Equation (18)). To obtain the sensitivity to random variation (ΔR), contributions from all other first-order deterministic sensitivities (e.g., ΔV, ΔT, ...) are subtracted from the total variance, isolating the portion attributable solely to the random component.

$$X_i \cdot T_X + Y_i \cdot (1 - T_X) \tag{18}$$

To accelerate the statistical max operation on GPU, we design a warp-efficient kernel that simultaneously processes 32 timing quantities, corresponding to a full warp of threads. As shown in Figure 8, each timing entry consists of eight elements: one nominal value and seven variation sensitivities. By packing four such entries (e.g., early/late and rise/fall) into a 32-element block, we can leverage a full warp of 32 threads to compute all directions in parallel.

Algorithm 4 presents the warp-parallel kernel for performing statistical `max` operations across timing arcs at each pin. Each thread determines its assigned work based on its `laneId`, where a warp is divided into four segments representing different mode and transition combinations (e.g., early-rise, late-fall). Within each segment, threads compute local timing quantities and participate in intra-segment reductions to obtain the intermediate result such as variances and ρ as described in Equation (16). The kernel takes the following inputs: the number of output pins in the current level (N); two input timing arrays `X` and `Y`, each storing arrival times or slews across early/late mode and rise/fall transitions. The output array `Z` stores the maximum values computed between the corresponding entries of `X` and `Y`. In addition, the array

Algorithm 4: Statistical Max Kernel

Input: N, X, Y
Input: *X_st, Y_st, Z_st*
Output: $Z = \max(X, Y)$
`// laneId: thread index in a GPU warp`
1 **parallel for each thread**
2 $rootLane \leftarrow \left\lfloor \frac{laneId}{8} \right\rfloor \times 8$
3 $mask \leftarrow$ `0xFF` $\ll rootLane$
4 $x = X[X\_st+laneId]$
5 $y = Y[Y\_st+laneId]$
6 $\sigma_X^2 \leftarrow$ **warp_reduce**$(mask, x^2)$
7 $\sigma_Y^2 \leftarrow$ **warp_reduce**$(mask, y^2)$
8 $\rho \leftarrow$ **warp_reduce**$(mask, xy)$
9 **if** $laneId == rootLane$ **then**
10 $\theta \leftarrow \sqrt{\sigma_X^2 + \sigma_Y^2 - 2\rho}$
11 $\alpha \leftarrow \frac{x-y}{\theta}$
12 $g_X \leftarrow \frac{1}{\sqrt{2\pi}}$`expf`$(\frac{-\alpha^2}{2})$
13 $T_X \leftarrow \frac{1}{2}(1 +$ `erff`$(\frac{\alpha}{\sqrt{2}}))$
14 **end**
15 **syncwarp**()
16 Broadcast θ, g_X, T_X, x, y from *rootLane*
`// Compute the nominal value`
17 **if** $\theta == 0$ **then** $z \leftarrow (x > y)\ ?\ x : y$
`// Compute the sensitivities`
18 **else**
19 **if** $laneId == rootLane$ **then**
20 $z \leftarrow x \cdot T_X + y(1 - T_X) + \theta \cdot \phi(\alpha)$
21 $\sigma_Z^2 \leftarrow (\sigma_X^2 + x^2) \cdot T_X + (\sigma_Y^2 + y^2)(1 - T_X) + (x + y) \cdot \theta \cdot g_X - z^2$
22 **end**
23 **else** $c \leftarrow x \cdot T_X + y(1 - T_X)$
24 **end**
25 $sum =$ **warp_reduce**$(mask, c^2)$
26 $Z[Z\_st + laneId] = z$
27 **if** $laneId == rootLane$ **then**
$Z[Z\_st + laneId + 7] = \sqrt{\sigma_c^2 - sum}$
28 **end**

pointers `X_st`, `Y_st`, and `Z_st` define the indices of the global memory for reading and writing these data.

Specifically, lines 6 uses the `warp_reduction` function to accumulate partial sums such as variances (σ_X^2, σ_Y^2) and correlation terms (ρ) across the corresponding *8-thread subgroup* within a GPU warp. These subgroups are aligned at warp lanes 0, 8, 16, and 24, each handling a distinct mode-direction pair (e.g., early-rise, late-fall). To implement this reduction efficiently, the kernel employs warp shuffle instructions, specifically `__shfl_down_sync`, which allow threads within a warp to directly read registers from other threads without using shared memory. Unlike traditional reductions that rely on global or shared memory with explicit synchronization, warp shuffle operations operate entirely in registers via hardware-supported lane communication, providing low latency and avoiding memory bottlenecks. For example, to compute the sum of eight values across lanes 0 through 7, each thread performs a sequence of shift-and-add operations in a binary-tree style:

$$\begin{aligned} val &= val + \_\_shfl\_down\_sync(mask, val, 4); \\ val &= val + \_\_shfl\_down\_sync(mask, val, 2); \\ val &= val + \_\_shfl\_down\_sync(mask, val, 1); \end{aligned} \tag{19}$$

At the end of this reduction, only the *first lane* of each 8-thread group (e.g., lane 0, 8, 16, or 24) holds the final sum, while the other lanes contain intermediate results that are discarded. This scheme ensures minimal instruction overhead and fully leverages warp-level SIMD execution. Once the reductions are complete, only the *root thread* of each group performs the final tightness probability and statistical max computation using Gaussian-based expressions. The resulting values (e.g., θ, T_X, g_X) are then broadcast to the other threads in the group using `__shfl_sync`, enabling synchronized access to computed constants across the group. This warp-level reduction scheme offers several benefits. First, it eliminates the need for shared memory, reducing resource contention. Second, all computation remains within a warp, avoiding the need for thread-block synchronization. In addition, each thread produces a scalar output, preserving SIMD efficiency while enabling parametric updates. Finally, lines 17–24 handle the statistical max and variance computation. The root thread calculates the expected value and variance using Equation (17), while other threads use the tightness probability model (18) to compute the sensitivity coefficients.

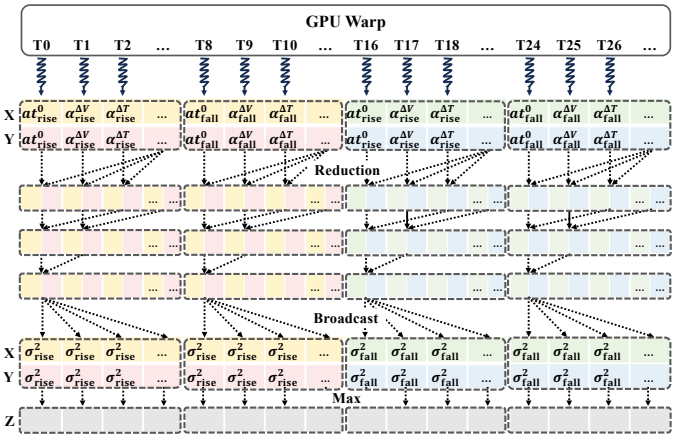


Fig. 8. An illustration of the statistical max operation kernel leveraging warp shuffle instructions and broadcast mechanisms for inter-thread communication.

This design minimizes memory traffic and fully utilizes GPU warp-level execution efficiency. Additionally, since the statistical min operation is structurally symmetric to max, it can be implemented by simply multiplying both inputs by -1, applying the same max kernel, and then negating the output.

IV. Experimental Results

A. Experimental Setup

We evaluate SSTA-X on the TAU Contest benchmarks and variation-aware cell library [16, 17], using the OpenTimer [25] front-end parser to read Verilog netlists, SPEF parasitics, Liberty libraries, and timing constraints. To ensure a fair and reproducible comparison, we implement a CPU-based Monte Carlo SSTA (MC-SSTA) as the golden reference and re-architect IITiMER [15] with OpenMP as the CPU-parallel baseline. MC-SSTA follows the same formulation as IITiMER and SSTA-X, but replaces Clark's approximation with direct Monte Carlo sampling. Slack mean and variance are estimated

TABLE I
PERFORMANCE COMPARISON AMONG THE GOLDEN REFERENCE MC-SSTA, THE BASELINE IITIMER, AND SSTA-X ON TAU CONTEST BENCHMARKS.

Circuit	#Gates	#Nets	#Pins	Runtime (ms)				Speedup of SSTA-X over			Error Rate
				MC-SSTA 20 CPUs	IITIMER 1 CPU	IITIMER 20 CPUs	**SSTA-X 1 GPU**	MC-SSTA 20 CPUs	IITIMER 1 CPU	IITIMER 20 CPUs	vs. MC-SSTA μ and σ^2 (%)
aes_core	22938	23199	66751	3483	129	26	**20**	174.15×	6.45×	1.30×	0.29 / 0.60
b19	255278	255300	782914	54989	1411	335	**134**	410.36×	10.50×	2.50×	0.38 / 0.96
cordic	45359	45393	127993	7286	226	70	**30**	242.86×	7.53×	2.33×	0.07 / 0.11
des_perf	138878	139112	371587	11403	580	115	**52**	219.28×	11.15×	2.21×	0.36 / 0.58
leon2	1616369	1616984	4328255	178620	6912	1667	**554**	322.41×	12.47×	3.01×	0.32 / 0.61
leon3mp	1247725	1247979	3376832	167752	5455	1487	**476**	352.42×	11.46×	3.10×	0.40 / 0.69
netcard	1496719	1498555	3999174	209258	6668	1799	**590**	354.67×	11.30×	3.04×	0.34 / 0.60
vga_lcd	139529	139635	397809	20233	674	157	**72**	281.01×	9.36×	2.18×	2.11 / 1.11
Average								327.96×	11.32×	2.81×	0.53 / 0.66

from 1000 samples using linear-regression-based sensitivity extraction. We do not compare against commercial tools such as Synopsys PrimeTime because their proprietary delay and variation models are not directly compatible with the TAU contest setting, and their additional timing infrastructure makes comparison difficult.

All experiments are conducted on an Ubuntu Linux 6.1.0-1022-oem x86_64 system with an Intel Core i5-13500 CPU (20 threads, 2.5 GHz), 128 GB RAM, and an NVIDIA RTX A4000 GPU. All programs are compiled with NVIDIA CUDA `nvcc` 12.7 and GNU GCC 12.3.0 under C++17 with `-O3` enabled. SSTA-X uses one CUDA stream for kernel execution and two for asynchronous CPU–GPU data transfer.

B. Overall Runtime Performance

Table I summarizes the benchmark characteristics and compares the overall runtime of MC-SSTA, IITiMER, and SSTA-X on eight TAU 2015 circuits, each with more than 10K gates. MC-SSTA and IITiMER use 20 CPU threads, while SSTA-X uses one GPU for timing kernels and 20 CPU threads for host-side preprocessing and levelization. Noted that the SSTA-X runtime reported in this table excludes CPU–GPU data transfer time, which is analyzed separately in Section IV-D. Across all benchmarks, SSTA-X consistently outperforms both CPU baselines, achieving an average speedup of 327.9× over MC-SSTA and 11.3× over IITiMER. Even against the 20-thread CPU baseline, SSTA-X still delivers up to 3.1× speedup, with the largest gains on `leon2`, `leon3mp`, and `netcard`. The speedup becomes more pronounced on million-gate designs, highlighting the effectiveness of the GPU kernels in exploiting the data parallelism of statistical timing propagation.

C. Scalability

Figure 9 compares the runtime scalability of IITiMER and SSTA-X on `leon2` and `netcard` as the CPU thread count increases from 1 to 20. IITiMER scales well up to 8 threads but saturates beyond 16 due to synchronization and contention overhead. SSTA-X consistently outperforms IITiMER across all thread counts by offloading timing propagation to the GPU, despite a mild slowdown beyond 16 threads from CPU-side levelization. Even with a single CPU thread, SSTA-X remains 2.7× faster on `leon2` and 2.6× faster on `netcard`, demonstrating the strong efficiency of SSTA-X.

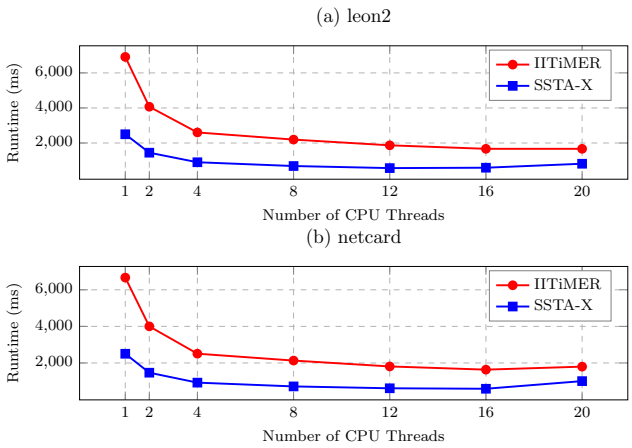


Fig. 9. Comparison of runtime scalability between IITiMER and SSTA-X at different CPU thread counts.

TABLE II
RUNTIME SCALING OF THE MAX AND CELL KERNELS WITH THE NUMBER OF SENSITIVITY DIMENSIONS (NDIM) ON A 100K-GATE LEVEL.

NDIM	Max (ms)	Cell (ms)	Total (ms)	Normalized
4	0.051	0.061	0.112	0.52×
8	0.102	0.114	0.216	1.00×
16	0.203	0.218	0.421	1.95×
32	0.417	0.433	0.850	3.94×

To evaluate scalability with respect to the number of variation sources, we measured the runtime of the two core CUDA kernels, while varying the number of sensitivity dimensions (NDIM). Table II reports the results on 100,000 gates. The runtime scales nearly linearly with NDIM. This trend is consistent with the linear growth in per-gate arithmetic cost and indicates that the proposed kernels remain efficient as the variation dimension increases.

D. Runtime Breakdown

Figure 10 shows the runtime breakdown of IITiMER (20 CPU threads) and SSTA-X on the largest benchmark, `leon2`. In IITiMER, levelization accounts for 38.2% of runtime due to its single-threaded implementation, whereas SSTA-X parallelizes this step and achieves a 1.6× speedup. For interconnect timing, SSTA-X combines parallel BFS RC-tree traversal with a vectorized GPU kernel, reducing runtime

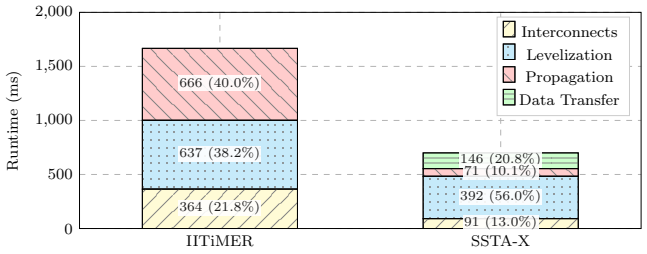


Fig. 10. Runtime breakdown of IITiMER and SSTA-X on `leon2`.

to 91 ms and delivering a $4\times$ speedup over IITiMER. For timing propagation, SSTA-X employs warp-efficient kernels for both forward and backward propagation, achieving a $9.4\times$ speedup. The figure also shows that CPU–GPU data transfer accounts for 20.8% of the total runtime of SSTA-X. Excluding CPU-side levelization, data transfer accounts for 52.6% of the remaining runtime, slightly exceeding the 47.4% spent on GPU kernel computation. Because the circuit topology and graph layout remain unchanged in incremental timing scenarios, this transfer overhead is effectively a one-time setup cost and can be amortized over repeated analyses.

E. Absolute Efficiency of GPU Acceleration

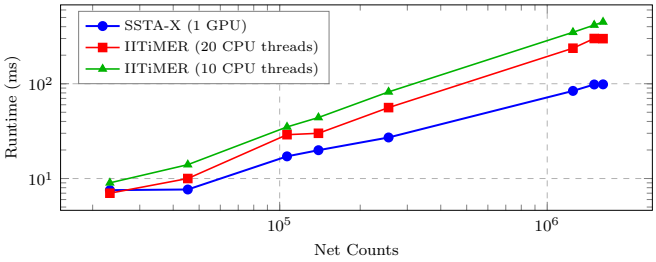


Fig. 11. Runtime comparison between our interconnect timing kernel and IITiMER (10 and 20 CPU threads) across different net counts.

We evaluate the absolute efficiency of SSTA-X across its three major kernel algorithms, *interconnect timing kernel*, *cell timing kernel*, and *statistical max kernel*, as shown in Figure 11, Figure 12, and Figure 13, respectively. In Figure 11, we compare the runtime of our interconnect timing kernel with IITiMER using 10 and 20 CPU threads. SSTA-X achieves over $4\times$ speedup over the 10-thread baseline and about $3\times$ over the 20-thread baseline, with the advantage increasing with net count. Once the design exceeds 50K nets, the GPU consistently outperforms all CPU configurations. This scalability is enabled by the flattened RC-tree kernel in Section III-B, which reduces workload imbalance through parallel, block-cooperative RC traversal and avoids the long-tail stalls of CPU DFS-based implementations.

Figure 12 shows the performance of our cell timing kernel. Using synthetically scaled circuits, SSTA-X consistently outperforms IITiMER, achieving up to $10\times$ speedup over the 10-thread CPU baseline and $8\times$ over the 20-thread version. This benefit comes from the warp-aware combinational kernel

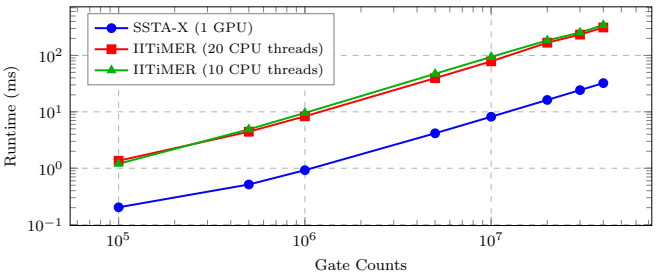


Fig. 12. Runtime comparison of our cell timing kernel and IITiMER (10 and 20 CPU threads) across different gate counts.

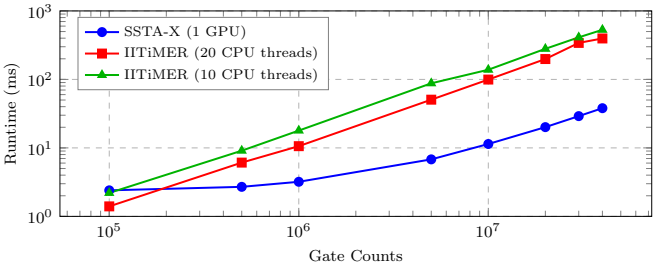


Fig. 13. Runtime comparison of our GPU-parallel statistical MAX kernel and IITiMER (10 and 20 CPU threads) across different gate counts.

in Section III-C, and the log–log trendlines indicate superior scaling as circuit size increases. Figure 13 evaluates our statistical max kernel. CPU and GPU runtimes are comparable on small circuits, but SSTA-X gains increasing advantage beyond 500K gates, reaching over $13\times$ speedup at 40M gates. This improvement is enabled by the warp-optimized max kernel in Section III-D, which reduces memory traffic and synchronization overhead and becomes more effective as circuit size grows. These results demonstrate that SSTA-X not only outperforms CPU-parallel baselines in runtime but also scales more effectively with increasing design complexity, making it well suited for large-scale statistical timing analysis.

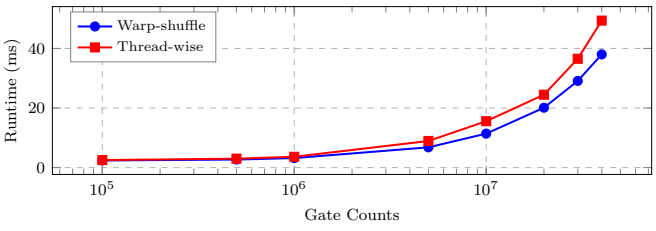


Fig. 14. Runtime comparison of the warp-shuffle statistical MAX kernel and a thread-wise baseline across different gate counts.

F. Ablation Study of GPU Optimizations

To evaluate the proposed statistical max kernel, we compare it with a thread-wise sequential baseline on synthetic circuits in Figure 14. The warp-shuffle kernel consistently outperforms the baseline, with speedup increasing from $1.05\times$ at 100k

gates to 1.30× at 40M gates. The advantage grows with circuit size because warp-shuffle reduction replaces sequential per-thread reduction with efficient intra-warp communication, reducing memory traffic and synchronization overhead. This result confirms the scalability and effectiveness of the proposed max kernel for large-scale timing analysis.

G. Accuracy and Robustness

TABLE III
SLACK-STATISTICS ERROR OF SSTA-X VS. MC-SSTA.

Circuit	#Levels	Mean (%)	σ (%)	95th (%)	99th (%)
leon2	154	0.06	317.82	0.86	1.23
leon3_mp	152	0.49	777.51	5.92	8.17
netcard	142	0.20	928.68	2.79	3.85

In terms of accuracy, the last column of Table I reports the average error in the mean and variance of pin arrival time relative to MC-SSTA. Since IITiMER and SSTA-X share the same underlying formulation, they exhibit identical accuracy. The average error is only 0.53% in the mean and 0.66% in the variance. To further assess the robustness of SSTA-X, we perform an additional validation of reconvergence-heavy tail behavior by selecting the top 200 endpoints with the largest numbers of max operations along their backward paths and comparing their slack distributions against MC-SSTA, as shown in Table III. The results show that mean slack remains accurate across all benchmarks, whereas tail metrics are more sensitive, with the 99th-percentile error consistently larger than the 95th-percentile error. This behavior is expected because FB-SSTA repeatedly applies analytical min/max operations and projects the resulting distributions back to the first-order Gaussian canonical form. As these approximations accumulate along reconvergent paths, they affect the distribution tails more strongly, where non-Gaussian effects are more pronounced.

TABLE IV
SLACK-STATISTICS ERROR UNDER ZERO- VS. FULL-CORRELATION.

Circuit	#Endpoints	σ (%)	95th (%)	99th (%)
leon2	149,466	29.36	8.13	8.28
leon3_mp	108,918	16.27	22.97	23.36
netcard	97,841	35.51	6.65	6.37

To evaluate sensitivity to correlation modeling, we perform a stress test by forcing ρ=0 in the analytical max operation and comparing the resulting endpoint distributions against the original full-ρ formulation on the three largest benchmarks, `leon2`, `leon3_mp`, and `netcard`. The results show that correlation tracking remains important. Forcing ρ=0 changes endpoint σ by 29.36%, 16.27%, and 35.51%, respectively. The corresponding 95th/99th-percentile slack changes are 8.13%/8.28%, 22.97%/23.36%, and 6.65%/6.37%. These results confirm that correlation materially affects statistical timing accuracy. The tightness-probability max formulation remains smooth when candidates are close, so small perturbations in ρ lead to bounded changes rather than abrupt branch switching.

To examine numerical stability, we modify the max kernel to record the residual variance used to recover the random component after sensitivity blending on the three largest benchmarks. Negative residual variance occurs in only 3.34%, 2.18%, and 3.74% of max operations, respectively, and its magnitude remains extremely small, with minimum values of -1.0×10^{-21}, -4.5×10^{-22}, and -7.0×10^{-22}. We clamp such cases to zero without renormalization and observe no numerical divergence or unstable sensitivities, indicating that the variance decomposition remains numerically stable even on the largest benchmarks.

TABLE V
NUMERICAL STABILITY OF THE MAX OPERATION.

Circuit	#Neg. Residuals	Neg. (%)	Min Residual
leon2	339,757	3.34	-1.0×10^{-21}
leon3_mp	183,307	2.18	-4.5×10^{-22}
netcard	377,867	3.74	-7.0×10^{-22}

H. Practical Considerations and Applicability

TABLE VI
POST-TIMING CPPR IMPACT ON SLACK.

Circuit	Slack σ Reduction (%)			3σ Slack Improvement (%)		
	Median	Mean	95th	Median	Mean	95th
leon2	3.95	22.79	100.00	0.07	0.26	1.09
leon3_mp	0.13	4.53	19.25	0.01	0.19	0.95
netcard	3.36	19.86	100.00	0.11	0.36	1.58

From a sign-off perspective, SSTA-X currently does not incorporate clock-path pessimism removal (CPPR). This omission does not affect the validity of our accuracy comparison because MC-SSTA uses the same timing graph, clock-tree model, and non-CPPR slack computation. To quantify the impact of excluding CPPR, we perform a post-timing CPPR analysis on the three largest benchmarks in Table VI. The median slack-σ reduction is 3.95%, 0.13%, and 3.36% for `leon2`, `leon3_mp`, and `netcard`, respectively, whereas the corresponding median 3σ-slack improvement is only 0.07%, 0.01%, and 0.11%. While the mean and 95th-percentile effects are larger, indicating that CPPR can substantially affect a small subset of endpoints, both SSTA-X and MC-SSTA use the same non-CPPR formulation in our experiments. Therefore, excluding CPPR does not materially bias their comparison.

TABLE VII
BACKWARD CONE COVERAGE OF THE TOP-K WORST-SLACK ENDPOINTS.

Circuit	Top-K	Unique Gates	% Circuit
aes_core	10	49	0.2
	50	10346	45.1
	100	13956	60.8
b19	100	9171	3.6
	1000	76353	29.9
	5000	203239	79.6

To clarify the regime in which full-graph propagation becomes more effective than path-based refinement, we perform a top-K endpoint fan-in cone analysis, as shown in Table VII. After ranking endpoints by worst slack, we measure the number of unique gates contained in the union of the backward fan-in cones of the top-K endpoints. The results show a clear

dependence on both circuit structure and K. For example, in `aes_core`, the combined cone already covers 45.1% of the circuit at K=50, whereas in `b19`, the top-100 endpoints cover only 3.6% of the graph, but the union grows to 79.6% at K=5000. This trend indicates that path-based refinement remains effective only when K is small and the selected endpoints cover a limited portion of the timing graph. As K increases and overlap expands, the benefit of restricting analysis to a subset of paths diminishes, making full-graph propagation increasingly favorable.

TABLE VIII
RUNTIME COMPARISON OF NLDM AND LINEAR DELAY MODELS.

#Gates	Delay Model (ms)		Max	Total (ms)		Slowdown
	NLDM	Linear	(ms)	NLDM	Linear	
100,000	1.30	0.09	0.10	1.31	0.19	6.8×
200,000	2.29	0.17	0.22	2.51	0.39	6.4×
400,000	4.48	0.32	0.48	4.96	0.80	6.2×
800,000	8.93	0.64	1.06	9.99	1.70	5.8×
1,600,000	17.73	1.29	2.11	19.84	3.40	5.8×

To evaluate whether the benefits of SSTA-X persist under other delay models, we implement a nonlinear delay model (NLDM) microbenchmark in which each gate performs bilinear interpolation on a 7×8 slew—load lookup table [17] and estimates sensitivities by finite differencing. Table VIII reports the runtime for different gate counts during a single-level forward propagation. Although NLDM increases the cost of cell-delay evaluation, the overall slowdown remains bounded at 5.8× to 6.8× because the statistical max kernel and the overall propagation flow remain unchanged. These results indicate that the core GPU acceleration in SSTA-X remains effective for lookup-table-based timing propagation.

V. CONCLUSION

In this article, we presented SSTA-X, a GPU-accelerated first-order block-based statistical static timing analysis framework for large designs. By introducing a topology-aware memory layout and warp-optimized GPU timing and statistical max kernels, SSTA-X enables efficient statistical timing propagation on GPU. Across the TAU Contest benchmarks, SSTA-X achieves up to a 3.1× speedup over IITiMER while maintaining high accuracy relative to MC-SSTA. Additional studies further validate the robustness and practical scope of SSTA-X. Reconvergence-heavy tail analysis shows that SSTA-X preserves mean slack accurately while maintaining bounded tail error under repeated analytical max approximation. Correlation stress tests confirm that correlation tracking remains important for statistical timing accuracy, and numerical diagnostics show stable variance decomposition even on the largest benchmarks. A post-timing CPPR analysis further indicates that excluding CPPR does not materially bias the comparison with MC-SSTA. In addition, a top-K cone analysis shows that SSTA-X is most advantageous in the large-scale full-graph regime, while an NLDM microbenchmark demonstrates that the proposed GPU acceleration remains effective under lookup-table-based delay evaluation.

A current limitation of SSTA-X is the absence of CPPR during slack computation, whose GPU acceleration remains challenging due to divergent clock-path dependencies and per-path correlation requirements. Another promising extension is a hybrid timing flow that combines GPU full-graph screening with path-based refinement. We view these directions as promising opportunities to further strengthen the practical applicability of SSTA-X. Overall, SSTA-X demonstrates that GPU acceleration is a scalable and effective direction for statistical timing analysis.

ACKNOWLEDGMENT

This project is supported by NSF grants 2349143, 2349144, 2349582, and 2349141. We acknowledge the use of AI tool (GPT-v5) to help polish the writing of this manuscript.

REFERENCES

[1] C. Forzan and D. Pandini, "Statistical static timing analysis: A survey," *Integration*, vol. 42, no. 3, pp. 409–435, 2009.

[2] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer, "Statistical timing analysis: From basic principles to state of the art," *IEEE TCAD*, vol. 27, no. 4, pp. 589–607, 2008.

[3] S. R. Nassif, "Modeling and forecasting of manufacturing variations," in *Proc. ACM/IEEE ASP-DAC*, 2001, pp. 145–150.

[4] V. Veetil, D. Sylvester, and D. Blaauw, "Efficient monte carlo based incremental statistical timing analysis," in *Proc. ACM/IEEE DAC*, 2008, pp. 676–681.

[5] A. Singhee, S. Singhal, and R. A. Rutenbar, "Practical, fast monte carlo statistical static timing analysis: Why and how," in *Proc. IEEE/ACM ICCAD*, 2008, pp. 190–195.

[6] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. ACM/IEEE ASP-DAC*, 2009, pp. 260–265.

[7] J. Cong, K. Gururaj, W. Jiang, B. Liu, K. Minkovich, B. Yuan, and Y. Zou, "Accelerating monte carlo based ssta using fpga," in *Proc. ACM/SIGDA ISFPGA*, 2010, pp. 111–114.

[8] C. Visweswariah, K. Ravindran, K. Kalafala, S. G. Walker, and S. Narayan, "First-order incremental block-based statistical timing analysis," in *Proc. ACM/IEEE DAC*, 2004, pp. 331–336.

[9] A. Agarwal, D. Blaauw, and V. Zolotov, "Statistical timing analysis for intra-die process variations with spatial correlations," in *Proc. IEEE/ACM ICCAD*, 2003, pp. 900–907.

[10] C. E. Clark, "The greatest of a finite set of random variables," *Oper. Res.*, vol. 9, no. 2, pp. 145–162, 1961.

[11] X. Huang, J. Lee, and P. Gupta, "Statistical static timing analysis in the UCLA timer," technical report, University of California, Los Angeles.

[12] L. Zhang, W. Chen, Y. Hu, and C. C.-P. Chen, "Statistical timing analysis with extended pseudo-canonical timing model," in *Proc. DATE*, 2005, pp. 952–957.

[13] M. Vijaykumar and V. Vasudevan, "Statistical static timing analysis using a skew-normal canonical delay model," in *Proc. DATE*, 2014, pp. 1–6.

[14] L. Zhang, W. Chen, Y. Hu, and C. C.-P. Chen, "Statistical static timing analysis with conditional linear MAX/MIN approximation and extended canonical timing model," *IEEE TCAD*, vol. 25, no. 6, pp. 1183–1191, 2006.
[15] N. Chandramoorthy, "IITiMER: TAU 2013 contest submission," 2013.
[16] D. Sinha, L. G. e Silva, J. Wang, S. Raghunathan, D. Netrabile, and A. Shebaita, "TAU 2013 variation aware timing analysis contest," in *Proc. ACM ISPD*, 2013, pp. 171–178.
[17] J. Hu, G. Schaeffer, and V. Garg, "TAU contest on incremental timing analysis," in *Proc. IEEE/ACM ICCAD*, 2015.
[18] V. Veetil, Y.-H. Chang, D. Sylvester, and D. Blaauw, "Efficient smart monte carlo based SSTA on graphics processing units with improved resource utilization," in *Proc. ACM/IEEE DAC*, 2010, pp. 793–798.
[19] J. Singh and S. S. Sapatnekar, "A scalable statistical static timing analyzer incorporating correlated non-gaussian and gaussian parameter variations," *IEEE TCAD*, vol. 27, no. 1, pp. 160–173, 2007.
[20] D. Sinha, H. Zhou, and N. V. Shenoy, "Advances in computation of the maximum of a set of gaussian random variables," *IEEE TCAD*, vol. 26, no. 8, pp. 1522–1533, 2007.
[21] Y.-C. Lu, Z. Guo, K. Kunal, R. Liang, and H. Ren, "Insta: An ultra-fast, differentiable, statistical static timing analysis engine for industrial physical design applications," in *Proc. ACM/IEEE DAC*, 2025, pp. 1–7.
[22] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of applied physics*, vol. 19, no. 1, pp. 55–63, 1948.
[23] J. Rubinstein, P. Penfield, and M. A. Horowitz, "Signal delay in RC tree networks," *IEEE TCAD*, vol. 2, no. 3, pp. 202–211, 1983.
[24] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *Proc. IEEE/ACM ICCAD*, 2020, pp. 1–9.
[25] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A new parallel incremental timing analysis engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2020.

Chih-Chun Chang received the B.S. and M.S. degrees in electrical engineering and computer science from Taiwan's National Tsing Hua University in 2017 and 2022, respectively. He is currently pursuing the Ph.D. degree in electrical and computer engineering at the University of Wisconsin–Madison.

His research focuses on GPU-accelerated algorithms for VLSI CAD. He previously interned at Cadence Design Systems, where he worked on GPU-accelerated timing analysis.

Mr. Chang received the Innovation Award at the 2023 IEEE HPEC Challenge, the Distinguished Paper Award at ACM UbiComp, and the championship in the NVIDIA Smart Embedded Robotics competition.

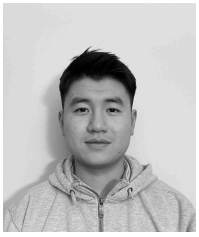
Yi-Hua Chung received the B.S. and M.S. degrees from National Taiwan University in 2021 and 2022, respectively. She is currently pursuing the Ph.D. degree in electrical and computer engineering at the University of Wisconsin–Madison, Madison.

Her research interests include GPU acceleration for electronic design automation, with a focus on partitioning algorithms for logic simulation and hybrid CPU–GPU systems.

Ms. Chung received the NTUEE-1975 Innovation and Entrepreneurship Fund Award, the 2022 Future Tech Award from the National Science and Technology Council, and the Best Paper Award at the 9th International Multi-Conference on Engineering and Technology Innovation in 2020.

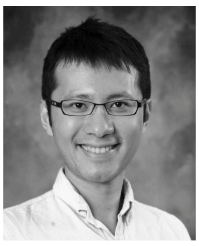
Wan Luan Lee received the B.S. degree in chemical engineering from Brigham Young University, and the Ph.D. degree in electrical and computer engineering from the University of Wisconsin–Madison. She is currently a Lead Software Engineer at Cadence Design Systems.

Her research interests include GPU-accelerated graph and hypergraph partitioning for scientific computing, data analytics, and VLSI design. Her work has appeared in venues including ACM TODAES, DAC, ASP-DAC, and Euro-Par.



Boyang Zhang received the B.S. degree in electronic and information engineering from China's South China University of Technology in 2020, and the M.S. degree in electrical and computer engineering from Rutgers University in 2021. He is currently pursuing the Ph.D. degree in electrical and computer engineering at the University of Wisconsin–Madison, Madison.

His research interests include task graph partitioning, parallel computing, and design automation. He has published at conferences including DAC, ICCAD, and ISPD.



Tsung-Wei Huang (Member, IEEE) received the B.S. and M.S. degrees from the Department of Computer Science in Taiwan's National Cheng Kung University in 2010 and 2011, respectively. He obtained his Ph.D. degree in electrical and computer engineering at the University of Illinois at Urbana-Champaign.

He is currently an Associate Professor at the ECE Department in the University of Wisconsin-Madison, Madison. His research focuses on building software systems to simplify the implementation of performance-critical applications, including CAD, machine learning, and quantum computing. His tools, such as OpenTimer and Taskflow, are widely used in industry and academia.

Dr. Huang has received several prestigious awards to recognize his contributions, including the ACM SIGDA Outstanding Ph.D. Dissertation Award, the NSF CAREER Award, the Humboldt Research Fellowship Award, the ACM SIGDA Outstanding New Faculty Award, ICCAD 10-Year Most Influential Paper Award (OpenTimer), DAC Under-40 Innovator Award, and the Vilas Early Career Investigator Award.