

GPU-Accelerated Gate-Level Time-Based Power Analysis via Event-Density-Aware Partitioning and Kernel Fusion

WEIHAO WANG, The Hong Kong University of Science and Technology - Guangzhou Campus, China
YIKANG OUYANG, The Hong Kong University of Science and Technology - Guangzhou Campus, China
HONGYUAN LIU, Stevens Institute of Technology, USA
YUZHE MA, The Hong Kong University of Science and Technology - Guangzhou Campus, China

Power analysis is crucial in modern chip design flow. Particularly, the time-based power analysis can provide fine-grained power consumption information to facilitate the diagnosis of power issues and guide power optimization accordingly. However, it may take tens of hours to conduct time-based power analysis on modern large-scale circuits, which greatly slows down the power optimization flow. In this paper, we present the first GPU-accelerated gate-level time-based power analysis framework. We propose a novel data structure to enable efficient state-dependent power retrieval. To accommodate the imbalanced event distribution across gates, we propose an event-density-aware partitioning strategy that allocates GPU threads based on gate event density. Finally, we fuse the power computation into a single kernel invocation to reduce redundant work in separate kernels. Experimental results show that our proposed framework achieves high accuracy while delivering up to 37.63× end-to-end speedup compared to multi-threaded Synopsys PrimeTime PX.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Hardware** → **Electronic design automation**; **Power estimation and optimization**.

Additional Key Words and Phrases: Time-based power analysis, GPU, parallelization strategies

ACM Reference Format:

Weihao Wang, Yikang Ouyang, Hongyuan Liu, and Yuzhe Ma. 2026. GPU-Accelerated Gate-Level Time-Based Power Analysis via Event-Density-Aware Partitioning and Kernel Fusion. *ACM Trans. Arch. Code Optim.* 1, 1 (January 2026), 26 pages. <https://doi.org/10.1145/3847118>

1 Introduction

Power has become a primary design constraint in modern very-large-scale integration (VLSI) systems, critically affecting performance, thermal management, and reliability. Consequently, accurate power analysis is indispensable throughout the design flow. While time-based power analysis using gate-level simulation offers the high temporal resolution required for accurate power reporting, it is computationally intensive. The increasing complexity of modern designs and long simulation traces exacerbate this challenge. For instance, analyzing a 288k-gate design with commercial tools can take several days [17]. This computational bottleneck hinders effective optimization, necessitating the acceleration of gate-level time-based power analysis.

New Paper, Not an Extension of a Conference Paper.

Authors' Contact Information: Weihao Wang, The Hong Kong University of Science and Technology - Guangzhou Campus, Microelectronics Thrust, Guangzhou, Guangdong, China, wwang103@connect.hkust-gz.edu.cn; Yikang Ouyang, The Hong Kong University of Science and Technology - Guangzhou Campus, Microelectronics Thrust, Guangzhou, Guangdong, China, yoyang929@connect.hkust-gz.edu.cn; Hongyuan Liu, Stevens Institute of Technology, the Department of Computer Science, Hoboken, NJ, USA, hliu96@stevens.edu; Yuzhe Ma, The Hong Kong University of Science and Technology - Guangzhou Campus, Microelectronics Thrust, Guangzhou, Guangdong, China, yuzhema@hkust-gz.edu.cn.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1544-3973/2026/1-ART

<https://doi.org/10.1145/3847118>

Prior research on power analysis can be broadly categorized into machine learning (ML)-based approaches and simulation-based approaches. Existing ML-based approaches [6, 19, 26, 28, 30] typically formulate power analysis as an *estimation* problem using register-transfer-level (RTL) activity proxies, and thus are not designed to explicitly perform event-driven, per-gate accumulation required for precise time-based power computation. Simulation-based approaches [10, 17] are more physically grounded, yet they often rely on *sampling or limited temporal coverage* to reduce overhead, which can miss short-duration glitches and transient peak power behaviors. In contrast, commercial tools such as Synopsys PrimeTime PX [21] support gate-level time-based power analysis and can generate complete and precise power reports (e.g., component breakdowns, waveforms, and peak values), but their runtime becomes prohibitive for large-scale designs. These limitations motivate accelerating gate-level time-based power analysis with high precision.

Given its inherent spatial and temporal parallelism, gate-level time-based power analysis is well suited for GPU acceleration, thereby alleviating computational bottlenecks and improving the overall design flow. However, fully exploiting this potential is non-trivial. We identify three key challenges to effectively leveraging GPUs for this workload: (1) *Inefficient state-dependent power retrieval*. Standard state-dependent power retrieval methods require two iterative list traversals per event. This incurs a substantial per-event cost, necessitating a more optimized data structure to handle high-frequency lookups. (2) *Imbalanced event distribution across gates*. The distribution of events per gate is highly skewed, and a globally fixed partitioning strategy is ineffective. Fine-grained partitioning incurs scheduling overhead for sparse gates with few events, whereas coarse-grained partitioning limits parallelism for dense gates with many events. Consequently, a workload-aware, per-gate granularity control is required. (3) *Redundant work in separate kernels*. Calculating dynamic and leakage power in separate kernels leads to significant redundancy, as both processes must independently perform costly gate-state reconstruction. Moreover, cycle-level information needed for glitch detection in dynamic power computation is already exposed during leakage power computation through cycle-level traversal. Therefore, a unified formulation is desirable to share gate-state reconstruction and reuse cycle-level information.

To address the above challenges, we propose a GPU-accelerated gate-level time-based power analysis framework. This framework supports efficient state-dependent power retrieval, uses adaptive per-gate partitioning to handle imbalanced event distribution across gates, and reduces kernel overhead through fused power computation. To the best of our knowledge, it is the *first* work that explores time-based power analysis on the GPU. This work¹ makes the following contributions:

- We identify three key challenges in effectively leveraging GPUs for time-based power computation. (Section 3)
- We propose a novel data structure that encodes state patterns into binary indices, enabling direct and efficient state-dependent power retrieval. (Section 4.1)
- We propose an event-density-aware, per-gate cycle partitioning strategy that efficiently distributes workload across GPU threads by leveraging each gate's event density and assigning threads accordingly. (Section 4.2)
- We propose a kernel fusion strategy that combines dynamic and leakage power computation within a single kernel invocation, thereby reducing redundant gate-state reconstruction and eliminating repeated cycle-level traversal of the same event streams. (Section 4.3)
- The proposed GPU-accelerated time-based power analysis framework leverages both spatial (gate-level) and temporal (cycle-level) parallelism to fully exploit the capabilities of GPUs. This framework achieves high accuracy and delivers an end-to-end speedup of up to 37.63× compared to multi-threaded PrimeTime PX. (Section 6)

¹The source code is publicly available at <https://github.com/whwang1996/GTPower>.

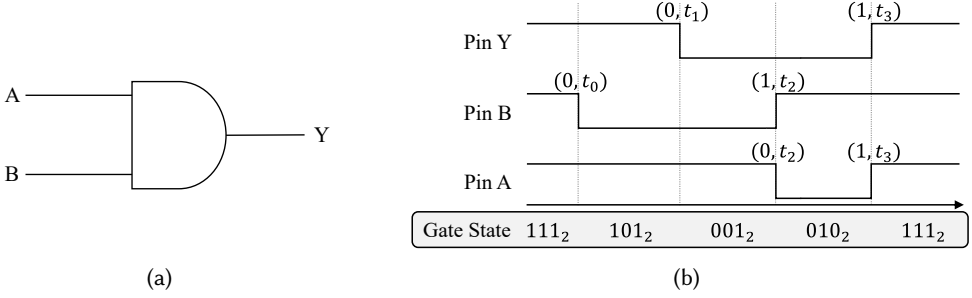


Fig. 1 Concept illustration using an AND gate. (a) Gate and its pins. (b) Events and gate states.

2 Preliminaries

2.1 Basic Concepts and Terminology

Fig. 1a illustrates an example gate with input pins A and B and output pin Y. Each pin carries a logic signal represented by a chronologically ordered event sequence. An *event* is a value–timestamp pair, indicating that the signal transitions to the corresponding logic value (0 or 1) at the associated time. For example, $(0, t_1)$ in Fig. 1b indicates the signal of pin Y transitions to 0 at t_1 . The *gate state* is the vector of logic values on all pins of the gate at a given time, and it changes whenever an event updates one of these pin values.

2.2 Power Models

2.2.1 Dynamic Power and Leakage Power. Circuit power is commonly decomposed into dynamic power and leakage power [25]. Dynamic power is dissipated when an event occurs and consists of internal power and switching power [24, 25]. Internal power is dissipated within a gate, and the corresponding event energy is obtained by interpolating a power lookup table (LUT) in the nonlinear delay model (NLDM) [4]. Note that the internal power can be state-dependent [16, 24]. As shown in Fig. 2, different gate states are associated with different internal power LUTs. Switching power is the power dissipation when the load capacitance at the gate output pin is charged or discharged. Leakage power is caused by intrinsic transistor leakage current, and its value persists between state-changing events. For gate g , the leakage value $p_{\text{leak}}^{(g)}$ is state-dependent [16, 24] and is retrieved from a pre-characterized array that maps gate states to leakage values (Fig. 3). Note that leakage power is always present; events only change the gate state and therefore modify $p_{\text{leak}}^{(g)}$.

2.2.2 Glitch Power. Beyond these fundamental power components, glitch power emerges as another critical consideration in power analysis. A glitch is an undesirable phenomenon caused by misaligned event timing, which leads to unnecessary events and consequently results in additional dynamic power consumption. Industry analysis indicates that glitches may contribute up to 40% of total power in modern digital circuits [23], making glitch-power analysis essential.

2.2.3 Power Waveform. Dynamic, leakage, and glitch power summarize component-level consumption but do not capture time-dependent fluctuations. The per-cycle power waveform provides this time-domain view and is crucial for diagnosing cycle-specific anomalies. This cycle-level visibility supports fine-grained root-cause analysis, e.g., attributing abnormal power to signal activity, input-state conditions, or transient events within a narrow window.

2.3 Power Analysis Modes

Power analysis is typically performed in averaged mode or time-based mode. The averaged mode is fast but reports only coarse-grained power over a specified interval. The time-based mode enables

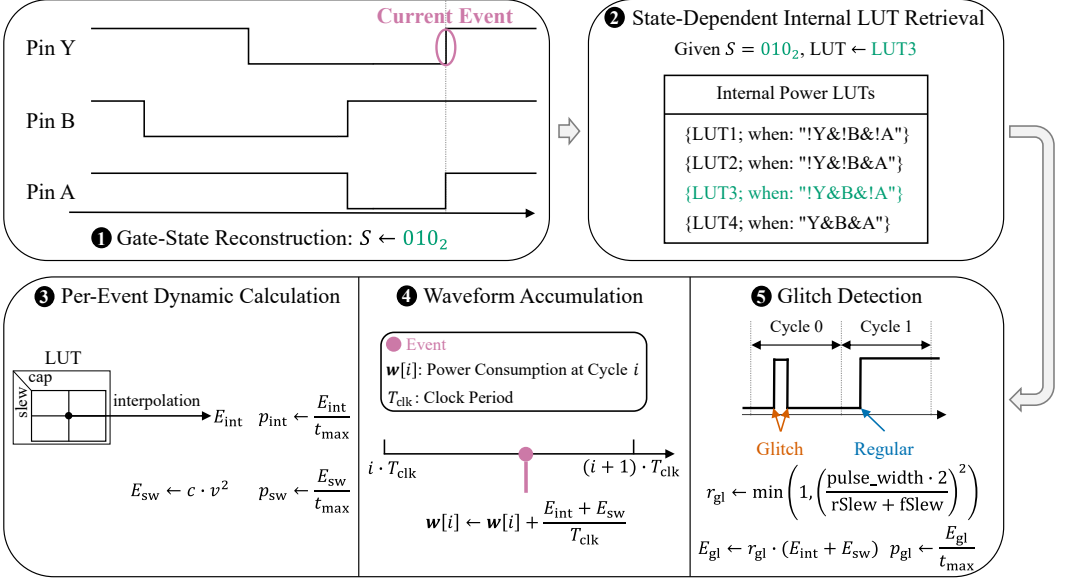


Fig. 2 Illustration of the per-event dynamic-power computation flow for Naive-D.

a comprehensive evaluation of power consumption by examining its temporal variations, thereby effectively capturing transient behaviors and dynamic characteristics of the circuit. In this mode, the power analysis tool computes power on an event-by-event basis, enabling detailed characterization of power consumption, including dynamic, leakage, and glitch components, as well as the power waveform. Although the time-based mode enables power analysis with fine temporal granularity, it incurs substantial computational overhead for large-scale designs due to the necessity of processing every transition event across all gates.

In a standard gate-level time-based power-analysis flow, design/model parsing, activity-file parsing, and static timing analysis (STA) provide the inputs required by event-driven power computation [24]. Design/model parsing reads the gate-level netlist, parasitics, constraints, and Liberty power models, including the internal power LUTs defined in the Liberty files; activity-file parsing reads per-pin transition events from the activity file; and STA provides the slew information required for LUT interpolation and glitch detection. These inputs are then used in the subsequent event-driven power-computation stage.

2.4 Time-Based Power Computation with a Naive GPU Implementation

Because no open-source time-based power-analysis implementation is available to our knowledge, we build a naive GPU implementation. Dynamic power occurs at discrete event times, whereas leakage power persists continuously; thus, they contribute to the waveform differently. Therefore, the naive implementation computes them separately using two GPU kernels: Naive-D for dynamic power and Naive-L for leakage power.

2.4.1 Naive-D: Event-Parallel Dynamic Power Kernel. In this kernel, each GPU thread is mapped to a single event on a specific pin of a gate and computes its dynamic-power contribution. For the assigned event, the computation first reconstructs the preceding gate state (Fig. 2 ①), scans the state-dependent array to retrieve the matching internal-power LUT (Fig. 2 ②), and computes the per-event internal and switching energy (Fig. 2 ③). The internal energy E_{int} is obtained by interpolating the retrieved LUT using input slew and output capacitance. And the switching energy

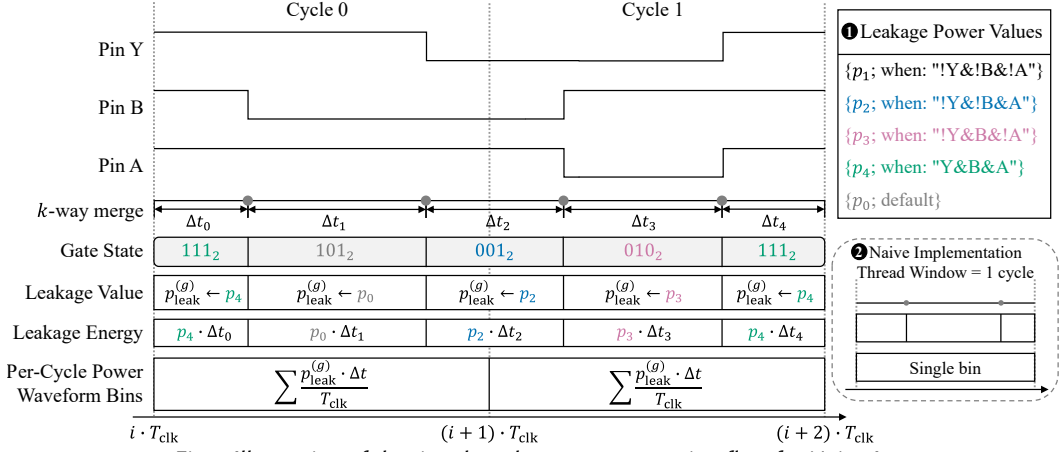


Fig. 3 Illustration of the time-based power computation flow for Naive-L.

E_{sw} of a single 0-to-1 event at an output pin is:

$$E_{\text{sw}} = c \cdot v^2, \quad (1)$$

where c is the total output load capacitance, including load pin capacitance and interconnect capacitance, and v is the power supply voltage [25].

Next, the event contribution is accumulated into the per-cycle waveform using atomic updates to combine contributions across threads (Fig. 2 ④). Finally, glitches are detected with a clock-cycle-based approach [24] (Fig. 2 ⑤). This method, which is widely used in commercial tools, provides an intuitive yet efficient solution. It detects glitch occurrences by counting the number of events within a single clock cycle. If the number of events exceeds one in a clock cycle, the events in that cycle are regarded as glitch events. For each glitch event, its dynamic energy is scaled into glitch energy E_{gl} by a ratio r_{gl} determined by the glitch pulse width and the rise/fall slews [24]:

$$r_{\text{gl}} = \min \left(1, \left(\frac{\text{pulse_width} \cdot 2}{r\text{Slew} + f\text{Slew}} \right)^2 \right), \quad (2)$$

where pulse_width is the glitch pulse width, rSlew and fSlew are rise slew and fall slew, respectively. After obtaining event energy E , its contribution p over the full analysis window is:

$$p = \frac{E}{t_{\text{max}}}, \quad (3)$$

where E denotes E_{int} , E_{sw} , or E_{gl} , and t_{max} is the maximum event time. Accumulating these per-event contributions gives the design-level internal, switching, and glitch power components.

2.4.2 Naive-L: Cycle-Parallel Leakage Power Kernel. In general, Naive-L partitions the temporal dimension into cycle windows of $n_{\text{cyc}}^{\text{fix}}$ contiguous cycles, where $n_{\text{cyc}}^{\text{fix}}$ is globally fixed across all gates. Each thread processes one window for one gate, computing leakage power and accumulating its per-cycle waveform contribution. The detailed procedure is depicted in Fig. 3. The thread first finds each pin's event-index range in its assigned window and reconstructs the gate state at the window start. It then performs an on-the-fly k -way merge over per-pin event streams. At each timestamp, it scans the state-dependent array (Fig. 3 ①) to retrieve the current $p_{\text{leak}}^{(g)}$, which remains constant over the preceding inter-event interval. The leakage energy of that interval is:

$$E_{\text{leak}} = p_{\text{leak}}^{(g)} \cdot \Delta t, \quad (4)$$

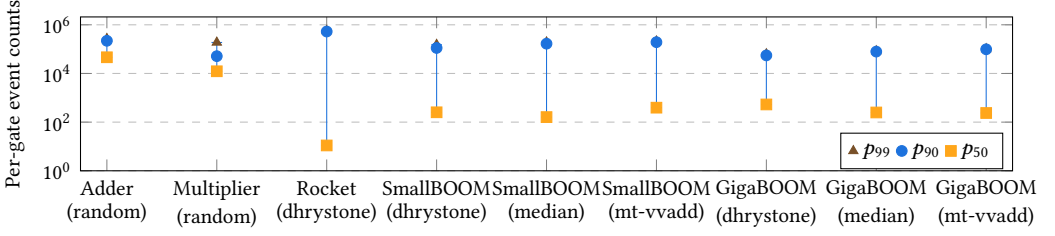


Fig. 4 Per-gate event-count percentiles (p_{50} , p_{90} , p_{99}) across nine datasets, shown on a logarithmic scale. The statistics of the evaluated designs are shown in TABLE 1, with the benchmark names indicated in parentheses.

where Δt represents the time interval between the two consecutive events.

For power waveform accumulation, each inter-event interval is split at clock-cycle boundaries, and the leakage contribution is added to every overlapped cycle bin in proportion to the overlap duration between the interval and that cycle.

After processing each timestamp, the gate state is updated by applying all events occurring at that timestamp, and the procedure repeats until the end of its assigned window.

The workload in Naive-L is two-fold: (i) the event-side cost, arising from the on-the-fly k -way merge over per-pin event streams and the associated gate-state updates; and (ii) the cycle-side cost, arising from updating the per-cycle power waveform bins for the thread's assigned cycles.

2.4.3 Parallel Granularity of Naive Kernels. In Naive-D, each thread is assigned exactly one event. Increasing the number of events handled by a single thread would require maintaining a consistent gate-state evolution across multiple pins, and thus a non-trivial kernel redesign. Consequently, the parallel granularity of Naive-D is inherently fixed to one event per thread. In contrast, Naive-L offers a tunable parallel parameter $n_{\text{cyc}}^{\text{fix}}$. For the naive implementation, we use the one-cycle-per-thread configuration ($n_{\text{cyc}}^{\text{fix}} = 1$), which represents the finest-grained setting (Fig. 3 ②). Together, these two kernels form the naive implementation for power computation in this work: *Naive-D adopts one-event-per-thread parallelism, whereas Naive-L employs one-cycle-per-thread parallelism.*

3 Characterization and Motivation

This section outlines three key challenges in leveraging GPUs for time-based power computation, derived from an analysis of the naive implementation introduced in Section 2.4.

3.1 Challenge #1: Inefficient State-Dependent Power Retrieval

As discussed in Section 2.2.1, the internal power and leakage power in a digital design can be state-dependent. One straightforward way to analyze state-dependent power consumption is to exhaustively traverse all state patterns and find the match. A significant limitation of this method is its computational inefficiency, as each event necessitates two separate matching procedures for leakage and internal power, respectively. Given a gate g with pin count $n_{\text{pin}}^{(g)}$ and $n_{\text{sd}}^{(g)}$ state-dependent power entries, each procedure runs in $O(n_{\text{pin}}^{(g)} \cdot n_{\text{sd}}^{(g)})$ time, incurring nontrivial runtime overhead. Therefore, an efficient data structure is required to reduce the overhead of retrieving state-dependent power information.

3.2 Challenge #2: Imbalanced Event Distribution Across Gates

In time-based power analysis, there are two dimensions of parallelism to leverage. Spatially, gate-level parallelism can be leveraged since the power is calculated separately for each gate. Temporally, event-level or cycle-level parallelism is also available, as the events or cycles of a gate are also

independent in the power computation. It is straightforward to leverage the spatial parallelism, as the power computations among different gates are independent. In contrast, temporal parallelism within each gate is harder to exploit efficiently, because highly skewed per-gate event counts make a global temporal granularity poorly balanced. Fig. 4 reports the p_{50} , p_{90} , and p_{99} (i.e., the 50th/90th/99th percentiles) of per-gate event counts across nine datasets. We observe that event counts at p_{90} and p_{99} are substantially larger than at p_{50} , reaching up to four orders of magnitude higher on Rocket’s dhrystone benchmark. This suggests a highly skewed distribution of event counts across gates. However, Naive-L adopts a uniform partitioning granularity across gates, with a fixed cycles-per-thread setting. As a result, the number of events processed per thread varies widely, leading to substantial variation in power computation workload across threads. This implies that a globally fixed cycles-per-thread parameter $n_{\text{cyc}}^{\text{fix}}$ cannot simultaneously accommodate all gates, and therefore fails to distribute the workload effectively. If $n_{\text{cyc}}^{\text{fix}}$ is too small, sparse gates with few events may still be allocated many threads. Many of these threads perform little useful work and then terminate, incurring substantial scheduling overhead. Conversely, if $n_{\text{cyc}}^{\text{fix}}$ is too large, the thread budget for dense gates with many events is reduced, preventing their heavier workloads from being effectively distributed and increasing the thread work depth. Therefore, we need fine-grained control over the parallel granularity of each gate according to its workload.

3.3 Challenge #3: Redundant Work in Separate Kernels

Both Naive-D and Naive-L perform gate-state reconstruction, leading to substantial redundant work when the two kernels are executed separately. Specifically, each thread must reconstruct the gate state at its assigned start time to enable state-dependent power retrieval. This is accomplished by performing a binary search on the event stream of each pin to identify the event index corresponding to its start time. Consider a gate g with $n_{\text{pin}}^{(g)}$ pins. Let k_i denote the event count at the i -th pin. The work required to find an event index corresponding to a specific time at pin i is $\log_2 k_i$, with each binary search iteration counting as one work unit. For a given thread, the work required to search the event indices associated with a specific time for gate g is:

$$W_{\text{search}}^{(g)} = \sum_{i=1}^{n_{\text{pin}}^{(g)}} \log_2 k_i \quad (5)$$

When allocating $N_{\text{thr}}^{(g)}$ threads to each gate g , the total work required to reconstruct the gate states of the entire design is:

$$\sum_{g \in G} N_{\text{thr}}^{(g)} \cdot W_{\text{search}}^{(g)} \quad (6)$$

Since two separate kernels are launched, the total number of threads $N_{\text{thr}}^{(g)}$ increases, thereby incurring non-negligible overhead for gate-state reconstruction.

In addition, Naive-D performs per-event glitch detection. This requires identifying the neighboring events within the same clock cycle via per-pin binary searches to locate cycle boundaries, incurring $O(\log_2 k_i)$ work per event on pin i . However, Naive-L already has access to cycle-level event count, because it explicitly traverses the event streams of all pins over time, thereby exposing the number of events that occur in each clock cycle.

Consequently, running Naive-D and Naive-L independently not only duplicates gate-state reconstruction but also recomputes glitch-related cycle statistics that could be reused, motivating a unified formulation that shares gate-state reconstruction and cycle-level information.

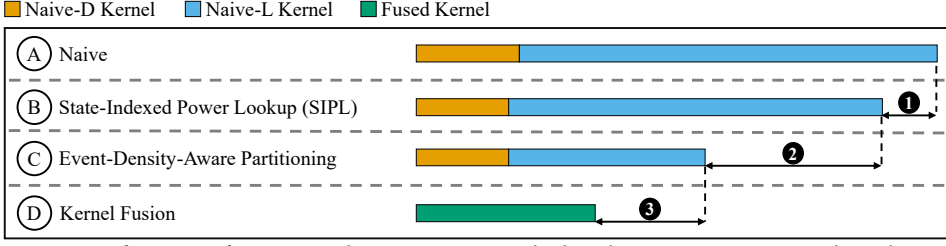


Fig. 5 Overview of a series of incremental optimizations applied to the power computation kernels: timelines from the naive implementation to the optimized version. Intended for illustrative purposes only; detailed experimental results are presented in Section 6.4.1.

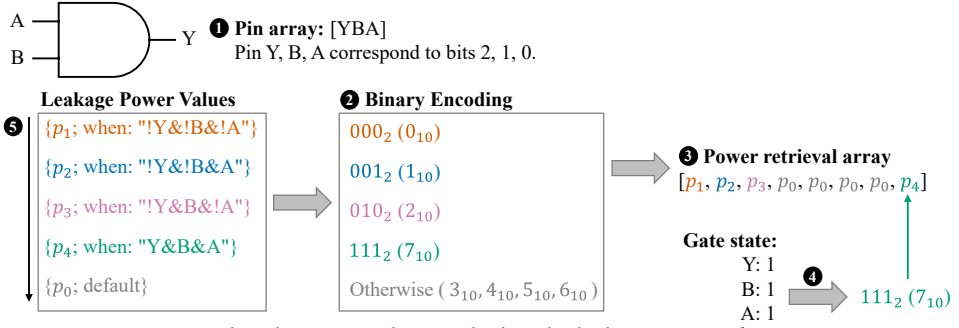


Fig. 6 State-Indexed Power Lookup applied to the leakage power of an AND gate.

4 Multi-Level GPU Optimizations

In this section, we introduce our GPU optimization techniques for addressing the challenges discussed in Section 3. For clarity in presenting our proposed methods, we have designed Fig. 5 as a multi-timeline visualization where each timeline graphically represents a distinct optimization technique. We begin with the naive implementation proposed in Section 2.4 (Fig. 5 (A)), where dynamic power is computed by the Naive-D kernel (one event per thread) and leakage power is computed by the Naive-L kernel (one cycle per thread).

4.1 Efficient State-Dependent Power Retrieval with State-Indexed Power Lookup

As discussed in Section 3.1, each event requires two $O(n_{\text{pin}}^{(g)} \cdot n_{\text{sd}}^{(g)})$ state-dependent power retrieval processes for leakage and internal power, respectively. To reduce this per-event overhead, we convert the traditional state-pattern traversal process into a *table lookup* using *State-Indexed Power Lookup* (SIPL). Specifically, during preprocessing, we process each cell in the Liberty file and organize its pins into a fixed-order array on the host side (Fig. 6 ①); this pin order is preserved in all subsequent computations. Each Liberty state pattern is then binary-encoded over this pin order, with each bit set according to the corresponding pin state (Fig. 6 ②). The resulting binary-encoded value is used as an integer index into a $2^{n_{\text{pin}}^{(g)}}$ -entry power retrieval array, where $n_{\text{pin}}^{(g)}$ denotes the cell pin count, and the corresponding power item is stored at the indexed entry (Fig. 6 ③).

Some Liberty state patterns specify only a subset of pins. Pins omitted from a pattern are treated as don't-care pins, meaning that the same power item applies regardless of their logic values. Thus, one state pattern may correspond to multiple SIPL indices. To populate the retrieval array, we process each Liberty state pattern by enumerating all binary assignments of its don't-care pins while keeping its specified pin values fixed. Each complete pin assignment is then binary-encoded into an array index, and the indexed entry is filled with the power item of the current state pattern.

During power computation, we encode the current gate state using the same pin order and use the encoded value as the array index, allowing us to retrieve the power item directly (Fig. 6 ④). The same construction is also applied to internal power LUTs, with the difference being that the elements in the power retrieval array are LUTs.

One important consideration of this method is that the number of pins can be too large. For example, the number of pins of an SRAM macro can be over one hundred. Directly mapping its state patterns into an array will result in the array size exceeding 2^{100} , which exceeds the memory capacity. To avoid excessive retrieval-array sizes, we use SIPL only for gates with at most 16 pins and fall back to direct state-pattern traversal for gates with more than 16 pins. In the fallback path (Fig. 6 ⑤), the original state patterns are scanned, each pattern is checked against the current gate state, and the power item associated with the matching pattern is retrieved. The choice of the 16-pin threshold is justified in Section 6.5.1. With this threshold, only 0.05% of gates invoke this fallback strategy in our experiments. By replacing state-pattern traversal with direct indexed access, SIPL reduces the complexity of state-dependent power retrieval from $O(n_{\text{pin}}^{(g)} \cdot n_{\text{sd}}^{(g)})$ to $O(n_{\text{pin}}^{(g)})$. As shown in Fig. 5, our SIPL data structure (B) reduces GPU kernel execution time by ① over A.

4.2 Adaptive Per-Gate Temporal Partitioning

In this subsection, we present an event-density-aware, per-gate partitioning scheme to accommodate the imbalanced event distribution across gates.

4.2.1 Event-Density-Aware Per-Gate Cycle Partitioning. As discussed in Section 3.2, the event counts across gates can be highly skewed. Under such inter-gate heterogeneity, using a globally fixed cycles-per-thread setting for Naive-L leads to systematic misallocation of parallelism. To accommodate the wide variation in per-gate event count, we adapt the temporal partitioning granularity on a *per-gate basis*. Specifically, we introduce a target event count per thread budget $n_{\text{evt,tgt}}$. For each gate g , we select the cycles-per-thread parameter $n_{\text{cyc}}^{(g)}$ such that each thread is expected to process approximately $n_{\text{evt,tgt}}$ events. Let N_{cyc} denote the total number of cycles and let $N_{\text{evt}}^{(g)}$ denote the total number of events associated with gate g . We partition the events of gate g along the temporal dimension by dividing the N_{cyc} cycles into contiguous windows of length $n_{\text{cyc}}^{(g)}$. Each thread processes the events of gate g whose timestamps fall within its assigned $n_{\text{cyc}}^{(g)}$ -cycle window. To estimate the expected per-thread workload, we define the average event density of gate g to be

$$\rho^{(g)} = \frac{N_{\text{evt}}^{(g)}}{N_{\text{cyc}}} \quad (7)$$

Treating $\rho^{(g)}$ as a coarse average event density over the full time range, a thread assigned $n_{\text{cyc}}^{(g)}$ cycles is expected to process approximately $\rho^{(g)} \cdot n_{\text{cyc}}^{(g)}$ events. Accordingly, a natural choice is $n_{\text{cyc}}^{(g)} \approx \frac{n_{\text{evt,tgt}}}{\rho^{(g)}}$, obtained by matching the expected per-thread event count to the target budget $n_{\text{evt,tgt}}$. However, this choice becomes ill-conditioned when $\rho^{(g)}$ is small and may lead to excessively large windows. As discussed in Section 2.4.2, the workload in Naive-L is inherently two-fold, comprising the event-side cost and the cycle-side cost. Therefore, for sparse gates with small $\rho^{(g)}$, choosing the parallel granularity solely based on event density can under-provision threads for the cycle-side work, leaving clock-cycle traversal and power waveform updates to only a small number of threads. To stabilize the granularity in the sparse regime, we introduce a parallelism floor $N_{\text{thr,floor}}$, which

induces a window size

$$n_{\text{cyc},\text{floor}} = \left\lceil \frac{N_{\text{cyc}}}{N_{\text{thr},\text{floor}}} \right\rceil. \quad (8)$$

We then convert this window size into a density-floor term

$$\rho_{\text{floor}} = \frac{n_{\text{evt},\text{tgt}}}{n_{\text{cyc},\text{floor}}}. \quad (9)$$

We combine the measured density and the density-floor term into an effective density

$$\rho_{\text{eff}}^{(g)} = \rho^{(g)} + \rho_{\text{floor}}. \quad (10)$$

Intuitively, ρ_{floor} acts as a density floor that prevents the inferred window size from growing too large when the measured density $\rho^{(g)}$ is small.

The final cycles-per-thread granularity is obtained by inverting $\rho_{\text{eff}}^{(g)}$ with integer rounding,

$$n_{\text{cyc}}^{(g)} = \text{round}\left(\frac{n_{\text{evt},\text{tgt}}}{\rho_{\text{eff}}^{(g)}}\right). \quad (11)$$

Given a thread with local index $\tau^{(g)}$ within gate g and clock period T_{clk} , the time window $[t_{\text{start}}, t_{\text{end}}]$ assigned to it is:

$$t_{\text{start}} = \tau^{(g)} \cdot n_{\text{cyc}}^{(g)} \cdot T_{\text{clk}} \quad (12)$$

$$t_{\text{end}} = (\tau^{(g)} + 1) \cdot n_{\text{cyc}}^{(g)} \cdot T_{\text{clk}} \quad (13)$$

4.2.2 Thread-Count Characterization. We now characterize the thread-count behavior induced by the proposed event-density-aware, per-gate cycle partitioning scheme.

The number of cycles per thread is

$$n_{\text{cyc}}^{(g)} = \text{round}\left(\frac{n_{\text{evt},\text{tgt}}}{\rho_{\text{eff}}^{(g)}}\right) \approx \frac{n_{\text{evt},\text{tgt}}}{\rho^{(g)} + \rho_{\text{floor}}}. \quad (14)$$

Therefore, the number of threads assigned to g is

$$N_{\text{thr}}^{(g)} = \left\lceil \frac{N_{\text{cyc}}}{n_{\text{cyc}}^{(g)}} \right\rceil \approx \frac{N_{\text{cyc}} \cdot (\rho^{(g)} + \rho_{\text{floor}})}{n_{\text{evt},\text{tgt}}} = \frac{N_{\text{cyc}} \cdot \rho^{(g)}}{n_{\text{evt},\text{tgt}}} + \frac{N_{\text{cyc}} \cdot \rho_{\text{floor}}}{n_{\text{evt},\text{tgt}}}. \quad (15)$$

Substituting $\rho^{(g)} = \frac{N_{\text{evt}}^{(g)}}{N_{\text{cyc}}}$ and $\rho_{\text{floor}} = \frac{n_{\text{evt},\text{tgt}}}{n_{\text{cyc},\text{floor}}}$ gives

$$N_{\text{thr}}^{(g)} \approx \frac{N_{\text{evt}}^{(g)}}{n_{\text{evt},\text{tgt}}} + \frac{N_{\text{cyc}}}{n_{\text{cyc},\text{floor}}}. \quad (16)$$

Moreover, because $n_{\text{cyc},\text{floor}} = \lceil N_{\text{cyc}}/N_{\text{thr},\text{floor}} \rceil$, we obtain

$$\frac{N_{\text{cyc}}}{n_{\text{cyc},\text{floor}}} \approx N_{\text{thr},\text{floor}}, \quad (17)$$

Consequently, the number of threads allocated to gate g can be approximated as

$$N_{\text{thr}}^{(g)} \approx \frac{N_{\text{evt}}^{(g)}}{n_{\text{evt},\text{tgt}}} + N_{\text{thr},\text{floor}}. \quad (18)$$

This additive form clarifies the behavior in two limiting regimes. When the workload is sparse ($N_{\text{evt}}^{(g)}$ is small and thus $\rho^{(g)}$ is small), the event-driven term $\frac{N_{\text{evt}}^{(g)}}{n_{\text{evt},\text{tgt}}}$ becomes negligible, and the thread

Algorithm 1 Fused Power Computation Kernel

Input: The current gate g , the local thread index within gate g , the cycles-per-thread granularity of gate g , the clock period T_{clk} , the maximum of all event times, the design-level per-cycle power waveform $\mathbf{w}$.

Output: Thread-local leakage, internal, switching, and glitch power values, along with their contributions to the design-level per-cycle power waveform.

```

1: Initialize  $\mathbf{p} \leftarrow \mathbf{0}$ 
2: Initialize  $t_{\text{start}}$  and  $t_{\text{end}}$ 
3: Initialize gate state  $S$  at  $t_{\text{start}}$ 
4: Initialize per-pin glitch counter  $\mathbf{c} \leftarrow \mathbf{0}$ 
5: for  $t_{\text{cur}} \leftarrow$  the next minimum unprocessed event timestamp in  $[t_{\text{start}}, t_{\text{end}})$  do
6:   Get leakage power value  $p_{\text{leak}}^{(g)}$  by  $S$ 
7:   Update leakage power  $p_{\text{leak}}$  and waveform  $\mathbf{w}$ 
8:   for all pin has event at  $t_{\text{cur}}$  of  $g$  do
9:     Get the internal power LUT by  $S$ 
10:    Calculate dynamic (internal and switching) energy of the current pin
11:     $r_{\text{gl}} \leftarrow \text{GlitchDetectUpdate}(g, \text{pin}, t_{\text{cur}}, \mathbf{c}, T_{\text{clk}})$ 
12:    if  $r_{\text{gl}} > 0$  then
13:      Scale the dynamic energy by  $r_{\text{gl}}$ 
14:      Update glitch power  $p_{\text{gl}}$ 
15:    end if
16:    Update internal power  $p_{\text{int}}$  and waveform  $\mathbf{w}$ 
17:    Update switching power  $p_{\text{sw}}$  and waveform  $\mathbf{w}$ 
18:  end for
19:  Update gate state  $S$  by events occurring at  $t_{\text{cur}}$ 
20: end for

```

count $N_{\text{thr}}^{(g)}$ is dominated by the parallelism floor $N_{\text{thr, floor}}$, thereby avoiding over-parallelization while still maintaining sufficient parallelism. Conversely, when the workload is dense ($N_{\text{evt}}^{(g)}$ is large and thus $\rho^{(g)}$ is large), the event-driven term $\frac{N_{\text{evt}}^{(g)}}{n_{\text{evt, tgt}}}$ dominates $N_{\text{thr}}^{(g)}$, yielding a finer cycle granularity and allocating more parallelism (larger $N_{\text{thr}}^{(g)}$) to gates with heavier workloads.

In summary, the proposed event-density-aware per-gate cycle partitioning scheme reallocates parallelism according to each gate's event density. It avoids over-parallelization for sparse gates while increasing parallelism when event processing dominates, thereby improving overall parallelism utilization and reducing the overhead associated with excessive parallelization. As demonstrated in Fig. 5, this approach (©) achieves ② reduction in kernel execution time compared to ③.

4.3 Kernel Fusion for Unified Power Computation

As discussed in Section 3.3, both Naive-D and Naive-L require each thread to reconstruct the gate state, resulting in redundant work when the kernels are executed separately. Moreover, Naive-L already exposes cycle-level event-count information that can be reused for glitch detection, making the corresponding work in Naive-D redundant as well. Motivated by these observations, we propose a *fused kernel* that shares gate state and cycle-level information. This design computes both dynamic and leakage power within a single kernel invocation, thereby reducing redundant work.

The resulting per-thread control flow is shown in Algorithm 1. Each thread is assigned a cycle window according to the per-gate partitioning scheme described in Section 4.2. A global array $\mathbf{w}$

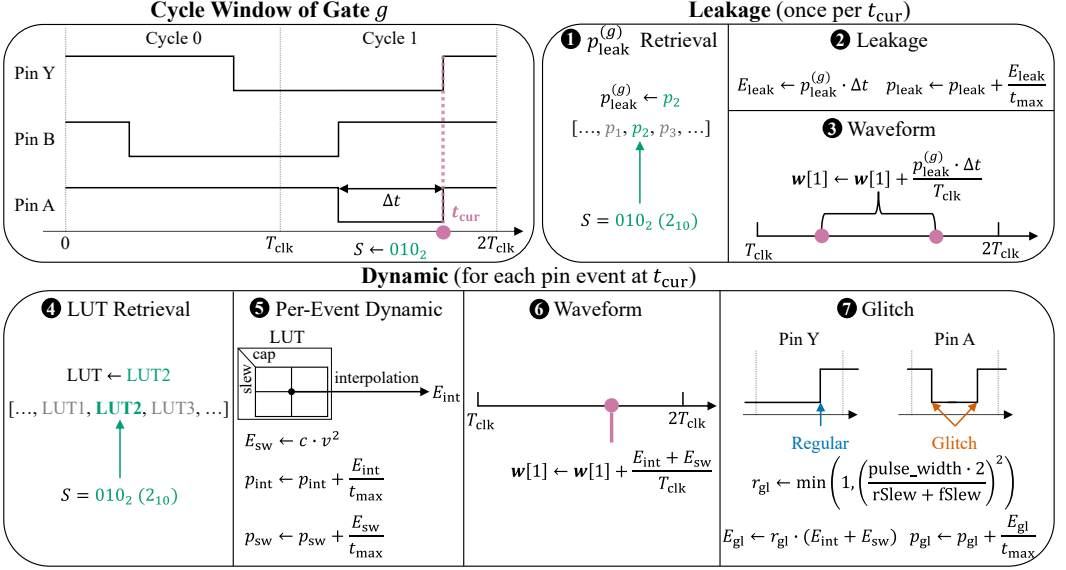


Fig. 7 Procedure for computing and accumulating leakage, internal, switching, and glitch power contributions at an event timestamp t_{cur} in the fused power computation kernel.

stores the design-level per-cycle power waveform. Each thread maintains a power results buffer $p = [p_{leak}, p_{int}, p_{sw}, p_{gl}]$ initialized to zeros, which accumulates the thread-local contributions of leakage, internal, switching, and glitch power before reduction (Algorithm 1). The time window of the current thread $[t_{start}, t_{end}]$ is derived using Equation (12) and Equation (13) (Algorithm 1). The gate state S at t_{start} and the per-pin glitch counter c are *initialized once and then reused throughout the thread's time window* (Algorithm 1–Algorithm 1). Here, c records the number of events on each pin within the current clock cycle. The outer for loop enumerates all event timestamps t_{cur} in chronological order via an on-the-fly k -way merge of the per-pin event streams. Each iteration performs the per-event-timestamp computation illustrated in Fig. 7 and terminates when no unprocessed event timestamps remain (Algorithm 1–Algorithm 1). The detailed power computation process will be presented in the rest of this subsection.

4.3.1 Leakage Power (Algorithm 1–Algorithm 1). For leakage power, the leakage power value $p_{leak}^{(g)}$ is retrieved based on the gate state S by our SIPL method (Fig. 7 ①). Subsequently, the leakage energy E_{leak} dissipated from the previous event to the current event is calculated using Equation (4). Then the leakage power is calculated using Equation (3) and added to p_{leak} (Fig. 7 ②). In addition, the leakage energy dissipation across cycles is cumulatively incorporated into w via a `atomicAdd` (Fig. 7 ③).

4.3.2 Dynamic Power (Algorithm 1–Algorithm 1). Since multiple pins of a gate may have events at the same timestamp, Algorithm 1 iterates over all pins with an event at t_{cur} and computes their dynamic power contributions. The calculation of dynamic power comprises two components: internal power and switching power. Internal power calculation requires retrieving the LUT associated with the gate state S , which is obtained via the SIPL-based retrieval method proposed in Section 4.1 (Fig. 7 ④). After retrieving the internal power LUT, the internal energy E_{int} is computed by interpolating the LUT using input slew and output capacitance, whereas the switching energy E_{sw} is calculated according to Equation (1). Following this, if the current event is not a glitch, the event's contribution to the corresponding dynamic-power component is calculated using

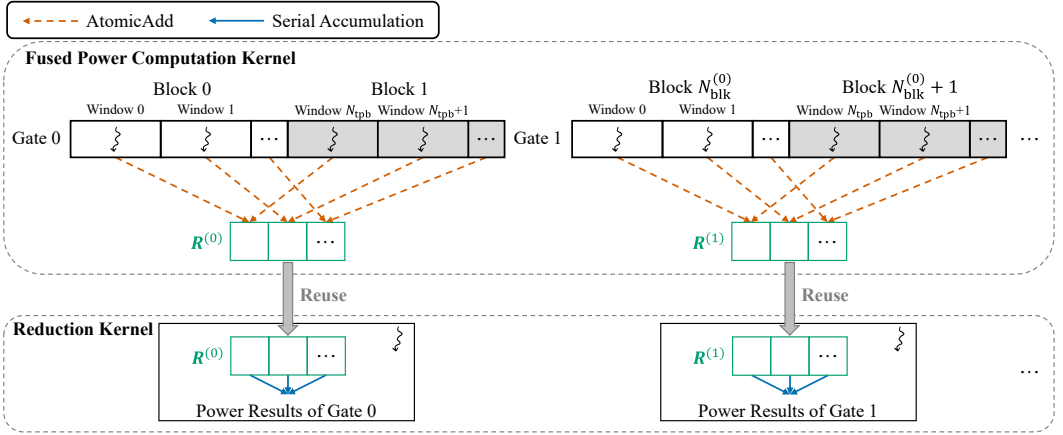


Fig. 8 Illustration of the mapping from cycle windows to threads and the two-phase power results reduction.

Equation (3) (Fig. 7 5). Additionally, the event's dynamic energy is accumulated into the power waveform $\mathbf{w}$ via `atomicAdd` in the clock cycle in which the event occurs, as illustrated in Fig. 7 6.

4.3.3 Glitch Power (Algorithm 1–Algorithm 1). `GlitchDetectUpdate(·)` (Algorithm 1) performs clock-cycle-based glitch detection by maintaining the per-pin event counter $\mathbf{c}$, following the method introduced in Section 2.4.1 (Fig. 7 7). If a pin has more than one event within the current clock cycle, the current event is treated as a glitch and a scaling ratio r_{gl} calculated using Equation (2) is returned; otherwise, -1 is returned to mark it as a regular one. After an event is marked as a glitch ($r_{gl} > 0$), its dynamic energy is scaled by the glitch scaling ratio r_{gl} . The glitch energy E_{gl} is the scaled dynamic energy, and the event's contribution to the glitch-power component is calculated using Equation (3).

As depicted in Fig. 5, by reusing the gate state S and the per-pin event counter $\mathbf{c}$ within each thread, kernel fusion (D) achieves 5 reduction in kernel execution time compared to C.

4.4 Thread Mapping and Two-Phase Power Results Reduction

As shown in Fig. 8, our GPU implementation employs one-dimensional thread organization for power computation, where each thread processes a continuous cycle window of its assigned gate. The number of threads allocated to each gate, denoted by $N_{thr}^{(g)}$, is determined by that gate's event density, as introduced in Section 4.2. Given a fixed number of threads per block N_{tpb} , the number of blocks assigned to each gate, $N_{blk}^{(g)}$, is derived from its thread allocation (i.e., $N_{blk}^{(g)} = \lceil N_{thr}^{(g)} / N_{tpb} \rceil$).

The GPU execution involves two distinct kernels: a fused power computation kernel and a reduction kernel. In the fused power computation kernel, each thread computes partial power results for its assigned gate, storing the results into a thread-local buffer $\mathbf{p} = [p_{leak}, p_{int}, p_{sw}, p_{gl}]$. These partial results need to be reduced to obtain the final per-gate power results. Allocating global memory privately for the intermediate results of each thread would impose excessive GPU memory overhead. To tackle this issue, we propose a two-phase reduction approach as illustrated in Fig. 8. In the fused power computation kernel, threads initially accumulate their outputs in a per-gate reduction buffer $\mathbf{R}^{(g)} \in \mathbb{R}^{N_{tpb} \times 4}$, where $N_{tpb} = \text{blockDim.x}$. This matrix, stored in global memory, is dimensioned to align with the thread block size. The four columns of $\mathbf{R}^{(g)}$ correspond to leakage, internal, switching, and glitch power components, respectively. Each thread locates its assigned position in $\mathbf{R}^{(g)}$ using block-local thread index (`threadIdx.x`), then performs thread-safe accumulation via `atomicAdd`. Following the completion of the fused power computation kernel, buffer $\mathbf{R}^{(g)}$ will be reused. A dedicated reduction kernel processes all gate buffers in parallel, with

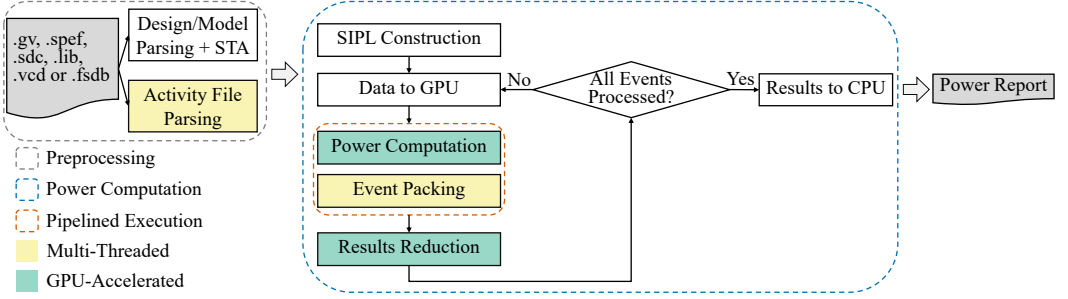


Fig. 9 Overview of the proposed GPU-accelerated gate-level time-based power analysis framework.

serial accumulation performed internally within each individual buffer, thereby yielding the final per-gate results. This design ensures conflict-free updates while achieving a balance between computational efficiency and GPU memory usage.

5 Framework Overview

After presenting our GPU-specific optimizations, we now introduce our proposed framework, as shown in Fig. 9. The framework begins with a two-step preprocessing phase on the CPU side. In the first step, the design/model parser reads the netlist (.gv), parasitics (.spef), constraints (.sdc), and Liberty (.lib) files to prepare gate-level physical parameters and Liberty power models, while the STA engine derives timing information. In the second step, the activity file (.fsdb or .vcd) is parsed using a multi-threaded parser that exploits signal-level parallelism. After parsing the activity file, we divide events into uniformly sized chunks to fit within limited GPU memory.

Following the preprocessing phase, the preprocessed design metadata (timing information, Liberty power models, gate-level physical parameters) and the chunked events are passed to the power-computation phase. This phase begins with two CPU-side setup steps: (1) Leakage power values and internal power LUTs are structured using the proposed SIPL data structure to facilitate efficient state-dependent power retrieval. (2) The first event chunk and the design metadata (timing/power/physical) are packed into contiguous buffers and transferred to GPU memory. The system then proceeds to perform power computation, operating iteratively to process the event chunks one by one. There are two main stages in each iteration. The first stage uses a pipelined CPU–GPU execution model that overlaps GPU-side power computation for the current chunk with CPU-side event packing for the next chunk. During the packing step, multiple CPU threads pack events from all signals in the next chunk into a contiguous buffer for efficient bulk transfer. The subsequent stage executes a GPU reduction kernel to reduce the power results for each gate. This process terminates upon completion of all event processing, with power computation results being transferred to the CPU. The system subsequently generates a comprehensive power analysis report, which includes both design-level and gate-level power consumption, along with the power waveform of the entire design. At the design level, leakage, internal, switching, and glitch power are obtained by summing the corresponding power contributions across all gates. The peak power is the maximum power value observed across all cycles in the power waveform w .

6 Experimental Results

6.1 Settings

System Configurations. We implement the framework with C++ and CUDA. The C++ code is compiled by GCC 11.4.0, and the CUDA code is compiled by NVCC 12.4, both with the -O3 optimization enabled. Unless otherwise stated, experiments are conducted on a CentOS 7.9 machine

Table 1 The design statistics.

Design	#Gates	#Pins	#Nets
Adder	1,815	5,220	2,457
Multiplier	13,267	42,523	15,439
Rocket	116,617	448,063	121,379
SmallBOOM	288,789	1,116,613	300,649
GigaBOOM	1,053,918	4,435,524	1,183,615

equipped with dual-socket AMD EPYC 7543 CPUs (2.8 GHz, 32 cores per socket, 64 cores total), 512 GB RAM, and an NVIDIA RTX 3090 GPU.

Power Analysis Configurations. For all designs, we use 128 threads per block, set the SIPL fallback pin threshold to 16, set the target event count per thread $n_{\text{evt,tgt}}$ to 8, and set the parallelism floor $N_{\text{thr,floor}}$ to 512; the rationale for these parameters is discussed in Section 6.5. The activity file is in .fsdb format, which is parsed using the library provided by Verdi [22].

Baseline. We adopt PrimeTime PX Version T-2022.03-SP1 [21] as our baseline, comparing our end-to-end runtime against it while treating its power analysis results as a golden reference. The measured end-to-end runtime uses the same accounting boundary for our framework and PrimeTime PX runs: design/model parsing, activity-file parsing, STA, and time-based power computation. The design/model parsing portion includes parsing the gate-level netlist, parasitics, constraints, and Liberty power models. For our GPU-accelerated framework, this boundary covers all stages shown in Fig. 9; in particular, it also accounts for GPU-specific implementation costs such as SIPL construction, CPU-side event packing, and CPU-GPU data transfer. To ensure a fair comparison, the glitch detection method is set to clock-cycle-based, and the interval of the power waveform is set to the clock cycle, which ensures consistency between the baseline and our implementation. PrimeTime PX is configured with its maximum supported thread count (16 CPU threads), and our framework is similarly configured with 16 CPU threads and 1 GPU. It should be noted that the baseline also performs temporal partitioning of the workload, distributing the activity file into discrete time windows containing an identical number of cycles. A comparison with existing open-source or academic tools is not provided, as, to our knowledge, none currently support full time-based power analysis reporting.

Experimental Designs. The designs used in our experiment, including two arithmetic circuits and three RISC-V system-on-chip (SoC) designs [3], are presented in TABLE 1, covering various design sizes. All the designs are synthesized using the TSMC N28 process. The largest design GigaBOOM contains more than one million gates and nets.

6.2 Power Analysis Results Compared with PrimeTime PX

6.2.1 Precision. We use PrimeTime PX power results as the reference P_{ref} and compute relative error as $\frac{|P - P_{\text{ref}}|}{P_{\text{ref}}}$. TABLE 2 lists the accuracy and performance comparison results between the baseline and our framework, together with the activity factor of each workload. It can be seen that the power computation error is very small. Specifically, the average errors of leakage, internal, and switching power are 0.12%, 0.06%, and 0.05%, respectively, demonstrating precise power computation. The average errors of glitch and peak power are 0.11% and 0.71%, respectively, which remain low. The activity-factor column also shows that higher-activity workloads generally produce larger dynamic-power components for the same design.

6.2.2 Fine-Grained Spatial and Temporal Comparison. We further validate the power analysis results with a fine-grained spatial and temporal comparison. On the one hand, despite the very small error of the power analysis results in TABLE 2, spatial accuracy is important for analyzing and optimizing power hotspots. To this end, we use Kendall's Tau [11] to evaluate the correlation

Table 2 Results comparison on precision and runtime. (Power Unit: *mW*)

Design	Benchmark	#Cycles	AF ¹	P_{leak}	Err. (%)	P_{int}	Err. (%)	P_{sw}	Err. (%)	P_{gl}	Err. (%)	P_{peak}	Err. (%)	RT-ref. (s) ²	RT (s) ³	Speedup
Adder	random	100,000	0.145	0.06	0.00	0.77	0.04	0.04	0.00	0.01	0.00	1.26	1.54	37.94	7.43	5.10
Multiplier	random	100,000	0.300	0.43	0.00	3.37	0.21	1.43	0.00	2.89	0.17	8.49	0.18	147.71	75.97	1.94
Rocket	dhrystone	529,602	0.047	5.13	0.06	27.29	0.04	0.37	0.00	0.65	0.05	64.49	1.05	1678.85	63.07	26.62
SmallBOOM	dhrystone	502,947	0.064	15.68	0.13	106.70	0.00	2.74	0.04	4.71	0.04	187.71	0.37	6872.54	332.35	20.68
SmallBOOM	median	167,541	0.058	15.75	0.32	102.90	0.00	1.75	0.06	3.68	0.14	187.46	0.55	2188.23	93.79	23.33
SmallBOOM	mt-vvadd	579,293	0.056	15.74	0.26	102.20	0.00	1.52	0.00	3.54	0.06	187.92	0.15	7767.76	206.44	37.63
GigaBOOM	dhrystone	359,397	0.041	44.55	0.11	265.50	0.08	6.60	0.08	9.47	0.16	467.22	0.76	20550.59	620.19	33.14
GigaBOOM	median	158,453	0.037	43.96	0.09	246.30	0.12	4.53	0.15	7.24	0.19	451.42	1.07	6041.12	311.58	19.39
GigaBOOM	mt-vvadd	541,712	0.036	43.95	0.11	243.20	0.08	3.98	0.15	6.79	0.21	459.32	0.75	20675.98	608.24	33.99

P_{peak} , P_{int} , P_{sw} , P_{gl} , and P_{peak} correspond to the leakage, internal, switching, glitch, and peak power of the whole design, respectively.

¹ Activity factor denotes the average number of 0-to-1 events per pin per simulated clock cycle.

² End-to-end runtime of PrimeTime PX using its maximum supported 16 threads.

³ End-to-end runtime of our framework using 16 threads + 1 GPU.

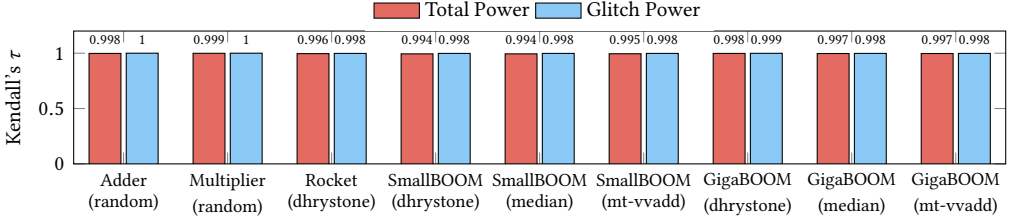


Fig. 10 Kendall's Tau of per-gate total power and glitch power ranking correlations between ours and baseline.

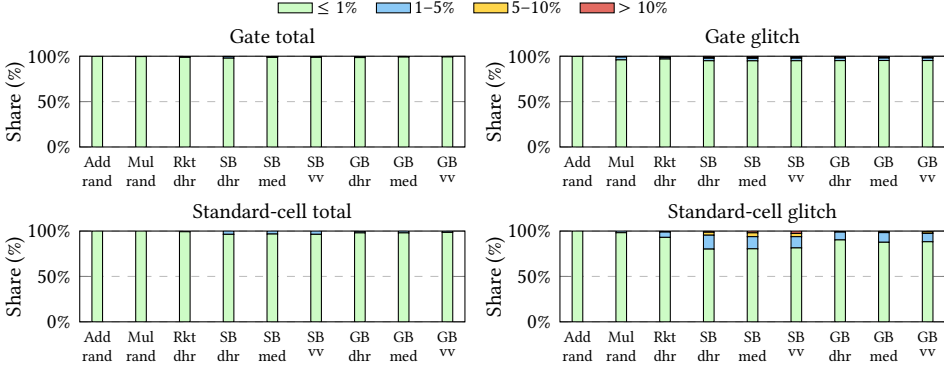


Fig. 11 Relative-error distributions of per-gate and standard-cell power values for total and glitch power. Each stacked bar shows the percentage of gates or standard-cell types in each error range.

in gate power dissipation rankings. As shown in Fig. 10, our framework achieves high ranking correlation, with Kendall's Tau coefficients no lower than 0.994 for total power and 0.998 for glitch power across all designs. Beyond ranking correlation, Fig. 11 reports the relative-error distributions for total and glitch power at the per-gate and standard-cell levels. Most gates and standard-cell types fall into the low-error ranges, showing that our framework closely matches the baseline in both ranking and absolute power values. In addition, the hotspot-overlap columns in TABLE 3 show that the hotspot overlap@1% remains high for both per-gate and standard-cell levels, confirming that our framework accurately identifies the highest-power gates and standard-cell types.

On the other hand, temporal power consumption information is important for diagnosing cycle-specific power issues. Therefore, we compare the power traces between PrimeTime PX and ours. For brevity, we select the smallest design Adder and the largest design GigaBOOM for power trace comparison, focusing on a segment with significant fluctuations for detailed analysis. As shown

Table 3 Hotspot overlap@1% and cycle-level power waveform accuracy compared with PrimeTime PX.

Design	Benchmark	Hotspot Overlap@1% (%)				Waveform Accuracy		
		Gate Total	Gate Glitch	Std. Total	Std. Glitch	RMSE (W)	NRMSE (%) ¹	Pearson Corr.
Adder	random	100.00	100.00	100.00	100.00	3.05×10^{-6}	0.614	1.000
Multiplier	random	100.00	100.00	100.00	100.00	7.98×10^{-6}	0.102	1.000
Rocket	dhrystone	99.74	99.29	100.00	100.00	1.58×10^{-4}	0.401	1.000
SmallBOOM	dhrystone	99.58	99.82	100.00	100.00	7.64×10^{-4}	0.704	0.999
SmallBOOM	median	99.03	99.37	100.00	100.00	4.26×10^{-4}	0.391	1.000
SmallBOOM	mt-vvadd	99.41	98.78	100.00	100.00	1.00×10^{-3}	0.914	0.997
GigaBOOM	dhrystone	99.49	99.50	100.00	100.00	1.47×10^{-3}	0.538	1.000
GigaBOOM	median	97.57	98.77	100.00	100.00	1.01×10^{-3}	0.391	0.999
GigaBOOM	mt-vvadd	99.01	98.89	100.00	100.00	2.18×10^{-3}	0.842	0.998

¹ The NRMSE is computed by normalizing the RMSE by the dynamic range of the corresponding reference waveform.

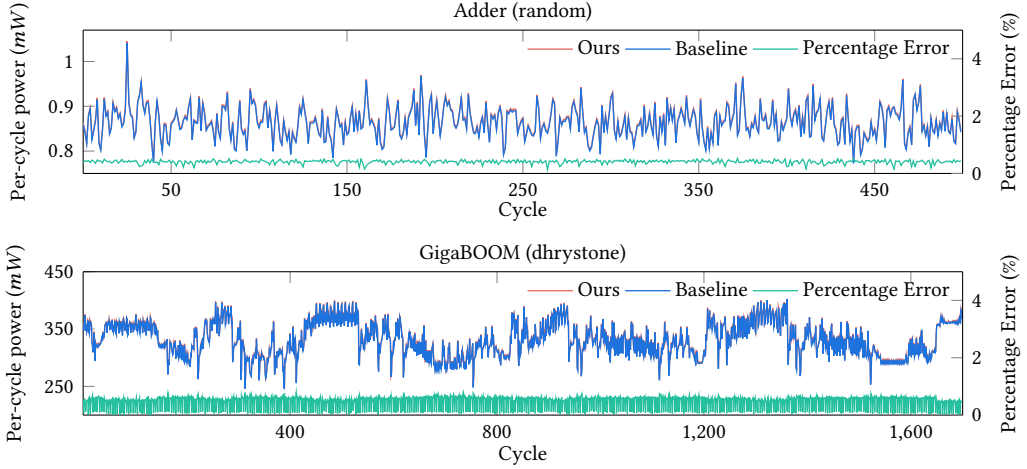


Fig. 12 Power traces comparison between ours and PrimeTime PX.

in Fig. 12, although there are some small deviations in values, the trends of our power traces are consistent with the baseline. The waveform-accuracy columns in TABLE 3 further confirm that the cycle-level power traces closely match PrimeTime PX, with low root mean square error (RMSE) and normalized RMSE (NRMSE), as well as high Pearson correlation across all benchmarks.

The strong consistency in both spatial and temporal dimensions validates the effectiveness of our framework in per-gate power optimization and cycle-level power analysis.

6.2.3 Speedup. In terms of efficiency, we compare the end-to-end runtime of the baseline and ours. It can be seen from TABLE 2 that we achieve a $1.94\times$ to $37.63\times$ end-to-end speedup compared to the multi-threaded PrimeTime PX baseline across various design sizes and benchmarks. Specifically, our framework achieves greater speedup on larger designs and longer benchmarks, as power computation typically requires more time for such cases. The runtime results demonstrate that our framework can perform ultra-fast and accurate time-based power analysis on large-scale designs.

6.3 Performance Analysis

We analyze framework performance through runtime breakdowns, CPU/GPU comparison, GPU memory footprint, and GPU bottleneck analysis. The CPU baseline in this section is a multi-threaded implementation that exploits gate-level parallelism using a publicly available thread pool [18]. Gates are shuffled for load balance and evenly assigned to worker threads by gate count; each worker computes event-driven power with thread-local accumulators before a final reduction.

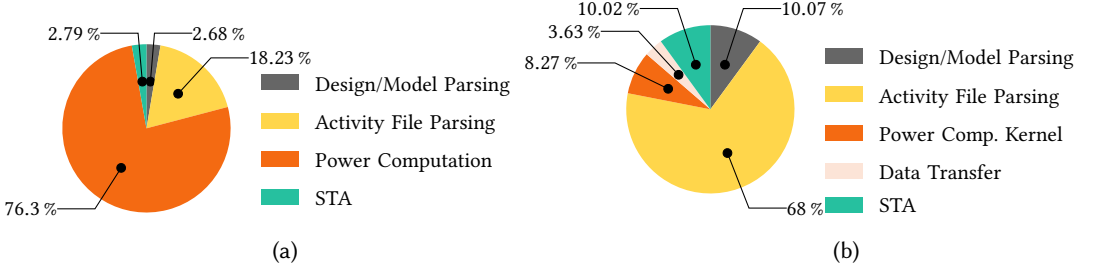


Fig. 13 The end-to-end runtime breakdown of the mt-vvadd benchmark on GigaBOOM. (a) CPU version (64 threads). (b) GPU version (64 host CPU threads and 1 GPU).

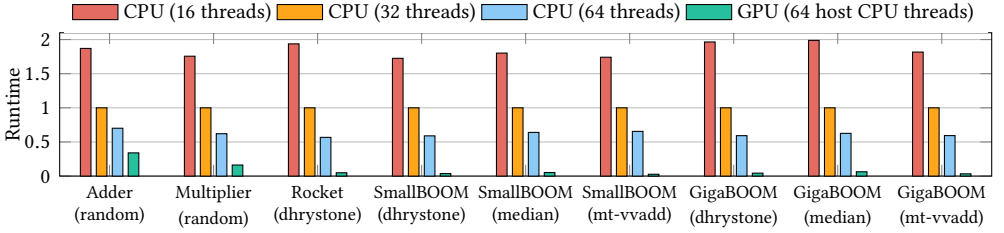


Fig. 14 Power-computation runtime of CPU implementations with 16, 32, and 64 threads and GPU with 64 host CPU threads, normalized to 32-thread CPU.

6.3.1 Runtime Breakdown. Fig. 13 presents the end-to-end runtime breakdown for both multi-threaded CPU (64 threads) and GPU (64 host CPU threads and 1 GPU) implementations on GigaBOOM with the mt-vvadd benchmark. For completeness, the GPU-accelerated flow also includes SIPL construction, CPU-side event packing, and a final-reduction kernel. Because event packing is overlapped with GPU execution and SIPL construction/final reduction account for less than 0.01%/0.22% of the end-to-end runtime, these components are omitted from the breakdown for readability. As demonstrated in Fig. 13a, the primary performance bottleneck in our multi-threaded CPU version is power computation, which accounts for 76.3% of the CPU end-to-end runtime. In contrast, Fig. 13b reveals that our GPU scheme effectively addresses this bottleneck: the GPU power-computation kernel and CPU-GPU data transfer together account for only 11.9%. With the power-computation bottleneck largely removed, the GPU end-to-end runtime is instead dominated by activity-file parsing, which accounts for 68.0%, shifting the remaining performance limit to the inherent file-parsing overhead in the end-to-end flow. For comparison, we also report the preprocessing and power-computation portions of PrimeTime PX on the same case. Its preprocessing consists of design/model parsing and STA, accounting for 0.6% of the end-to-end runtime. Because PrimeTime PX reads activity data during time-based power computation [24], this interleaved activity parsing is included in the power-computation category, which accounts for 99.4%.

6.3.2 CPU/GPU Comparison. Fig. 14 compares the power-computation runtime of our multi-threaded CPU implementation under 16, 32, and 64 CPU threads, together with the GPU implementation using 64 host-side CPU threads. In the GPU implementation, these host-side CPU threads are used for CPU-side work such as event packing. The measured runtime incorporates all steps within the power computation phase depicted in Fig. 9, excluding preprocessing. The results show that increasing the CPU thread count improves performance, but the gains remain sublinear: increasing the CPU thread count from 16 to 32 and from 32 to 64 achieves geometric-mean speedups of 1.84 $\times$ and 1.61 $\times$, respectively. This sublinear scaling is mainly due to the highly skewed per-gate event-count distribution, as shown in Fig. 4. In comparison, the GPU implementation with 64 host-side

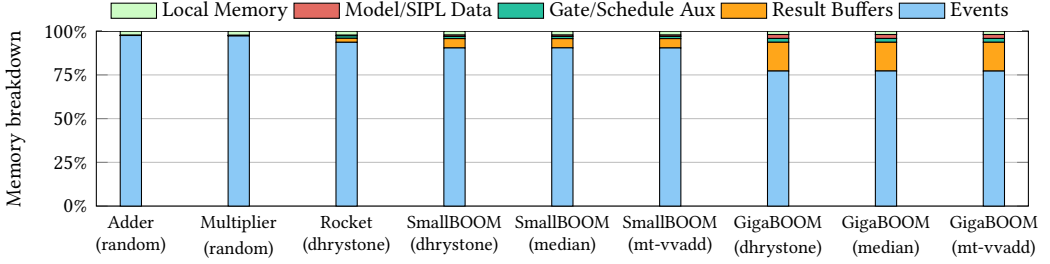


Fig. 15 Breakdown of peak GPU memory across benchmark designs.

Table 4 Nsight Compute profiling metrics of the fused power computation kernel.

Design	Benchmark	Memory/Compute Throughput (%)	DRAM Memory Throughput (Gbyte/second)	Achieved Occupancy ¹ (%)	Eligible/Issued Warps warps/sched.	Stall Share Long Scoreboard/Wait (%)	L1/L2 Hit Rate (%)
Adder	random	30.06/38.10	273.89	37.41	0.50/0.38	49.42/23.03	71.23/70.80
Multiplier	random	25.65/36.75	233.73	36.46	0.48/0.37	49.44/23.07	72.89/62.36
Rocket	dhrystone	43.26/31.38	385.42	37.23	0.40/0.31	59.60/19.22	53.98/65.73
SmallBOOM	dhrystone	30.26/35.62	244.95	37.40	0.47/0.35	53.99/21.59	61.75/75.14
SmallBOOM	median	37.83/34.81	289.82	37.96	0.47/0.34	55.39/20.65	57.59/76.85
SmallBOOM	mt-vvadd	40.94/34.15	313.46	39.25	0.46/0.34	56.88/19.65	56.27/76.89
GigaBOOM	dhrystone	35.11/34.55	208.21	37.67	0.47/0.34	54.22/20.78	60.20/81.84
GigaBOOM	median	40.99/33.35	215.30	37.67	0.45/0.34	55.47/19.99	55.22/85.56
GigaBOOM	mt-vvadd	45.01/32.46	223.23	38.90	0.44/0.32	57.34/18.58	52.30/86.35
Median		37.83/34.55	244.95	37.67	0.47/0.34	55.39/20.65	57.59/76.85

For benchmarks with multiple kernel invocations, profiling metrics are aggregated using kernel-duration-weighted averages.

¹ The theoretical maximum occupancy is 41.67%.

CPU threads achieves a 10.10× geometric-mean speedup over the 64-thread CPU implementation, effectively removing the primary power-computation bottleneck in time-based power analysis.

6.3.3 GPU Memory Footprint. We also analyze the GPU memory footprint to quantify the memory overhead of the framework. The measured peak GPU memory ranges from 12.22 GB to 15.44 GB across the evaluated benchmarks. As shown in Fig. 15, event storage is the dominant component, accounting for 77.2%–97.5% of peak memory, while result buffers become the second-largest component on larger designs and account for up to 16.5%. In contrast, model/SIPL data accounts for at most 2.3% of peak memory, indicating that SIPL does not change memory scalability of the flow.

6.3.4 GPU Kernel Bottleneck Characterization. To characterize the fused kernel, we collect Nsight Compute profiling metrics across the evaluated benchmarks. TABLE 4 shows that the fused kernel has low memory-throughput utilization (25.65%–45.01%) and low compute-throughput utilization (31.38%–38.10%), indicating that it is not primarily limited by memory bandwidth or arithmetic throughput. Instead, eligible warps per scheduler remain low, ranging from 0.40 to 0.50, and Long Scoreboard accounts for 49.42%–59.60% of all issue-stall reasons. These results indicate that the kernel is mainly limited by low eligible-warp availability, primarily due to long-latency dependency stalls associated with global-memory loads of event data and power-model metadata.

Because waveform-bin atomic updates may become a bottleneck under high switching activity, we run a stress test on GigaBOOM using 100,000-cycle synthetic high-activity waveforms with 1, 2, 4, and 8 toggles per cycle. Compared with a no-atomic variant that replaces `atomicAdd` with regular global-memory writes, waveform-bin atomic updates introduce only 2.03%–4.19% relative overhead. This limited overhead is primarily due to the low eligible-warp availability observed in profiling, which limits the number of warps issuing atomic updates concurrently. Moreover, practical activity traces often span hundreds of thousands of cycles, allowing concurrent updates to be spread across

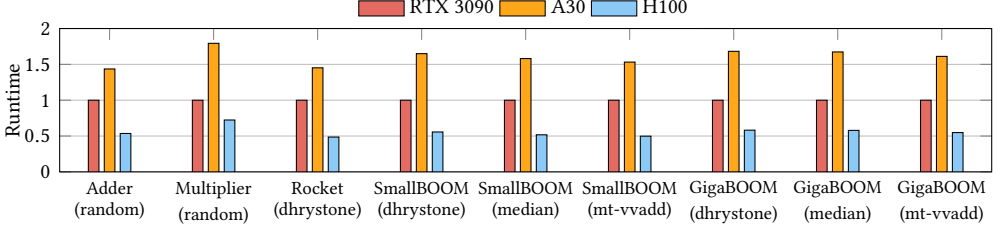


Fig. 16 Fused kernel runtime on RTX 3090, A30, and H100, normalized to RTX 3090.

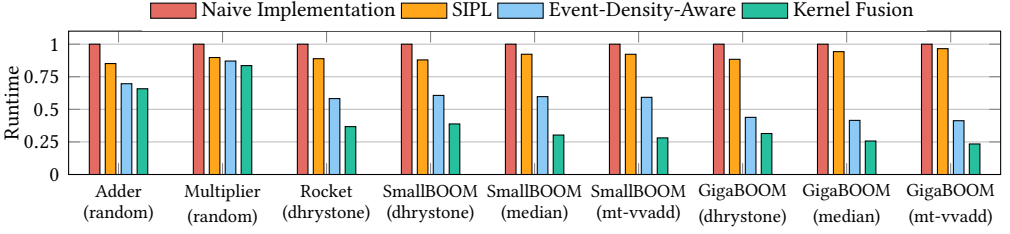


Fig. 17 Power computation kernel runtime under incremental optimizations. Normalized to "naive".

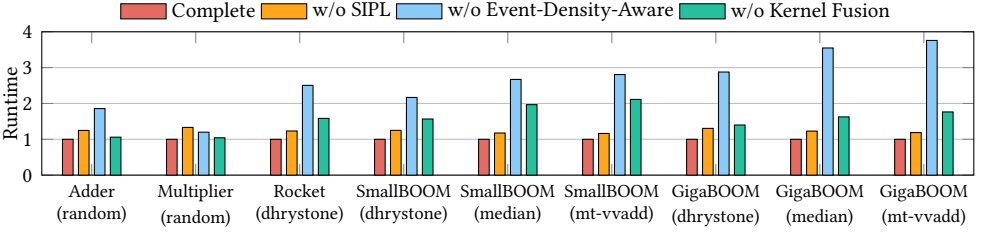


Fig. 18 Power computation kernel runtime under leave-one-out ablations. Normalized to the complete method.

many waveform bins and further limiting simultaneous conflicts on the same bin. These results indicate that waveform-bin atomic updates have limited impact on fused-kernel runtime.

6.3.5 Cross-GPU Platform Evaluation. To evaluate the portability of the proposed framework, we additionally measure the execution time of the fused power computation kernel on two other GPU platforms: an NVIDIA A30 GPU with 24 GB memory hosted by an Intel Xeon 6348 CPU, and an NVIDIA H100 GPU with 80 GB memory hosted by an Intel Xeon 8468 CPU. As shown in Fig. 16, A30 is slower than RTX 3090, with normalized runtime ranging from $1.44\times$ to $1.79\times$ (geometric mean: $1.60\times$), while H100 is faster, with normalized runtime ranging from $0.49\times$ to $0.72\times$ (geometric mean: $0.56\times$). These results confirm that the proposed framework is not limited to a single GPU architecture. The performance trend is consistent with the relative capability of the platforms, but the measured runtime does not scale simply with peak compute throughput or peak memory bandwidth. Together with the profiling results in TABLE 4, this non-linear scaling further suggests that the fused kernel is mainly affected by long-latency dependencies and warp readiness.

6.4 Optimization Strategies Evaluation

6.4.1 Incremental and Ablation Evaluation. To evaluate the effectiveness of our proposed optimizations in reducing the execution time of the power computation kernel, we build three incremental versions of the naive implementation, adding one optimization at a time to isolate and quantify each contribution. In the naive implementation, the Naive-L kernel sets the global cycle count per thread $n_{\text{cyc}}^{\text{fix}}$ to 1. The evaluation results presented in Fig. 17 reveal two key findings: First, the SIPL data

Table 5 Register usage, occupancy, and runtime sensitivity of the power computation kernel under fusion and occupancy-control variants.

Configuration	Registers per Thread	Theoretical Occupancy (%)	Achieved Occupancy (%) ¹	Normalized Kernel Time ²
Fusion	89	41.67	37.67	1.00
Fusion + Occ. 75%	56	75.00	66.53	1.60
Fusion + Occ. 100%	40	100.00	89.74	2.09
No Fusion	68/40	58.33/100.00	56.50/88.49	1.52

For the no-fusion variant, paired values correspond to the dynamic and leakage power computation kernels, respectively.

¹ Achieved occupancy is first aggregated per benchmark using kernel-duration-weighted averaging. The table reports the median across the nine benchmarks.

² Normalized kernel time is first computed for each benchmark relative to the fusion baseline and then aggregated using the geometric mean across the nine benchmarks. For the no-fusion variant, kernel time is the sum of the dynamic and leakage power computation kernel times.

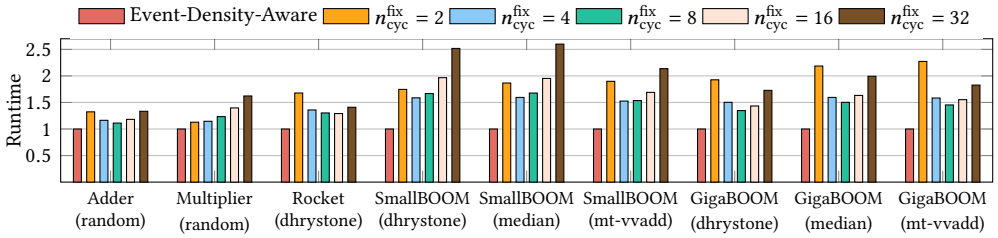


Fig. 19 Fused power computation kernel execution time with different global cycle count per thread (n_{cyc}^{fix}). Normalized against event-density-aware per-gate partitioning with $n_{evt,tgt} = 8$.

structure, event-density-aware per-gate cycle partitioning, and kernel fusion versions bring 9.45%, 43.67%, and 63.12% geometric mean runtime reductions compared to the naive implementation. Second, the event-density-aware per-gate cycle partitioning achieves higher speedups on the other designs than on Adder and Multiplier, because, as shown in Fig. 4, their per-gate event count distributions are more skewed than those of Adder and Multiplier.

To clarify each optimization's role in the complete implementation, we conduct a leave-one-out ablation study, as shown in Fig. 18. Removing event-density-aware partitioning causes the largest slowdown, increasing kernel runtime by 147.4% in geometric mean and up to 275.9%. Removing kernel fusion and SIPL increases geometric-mean runtime by 52.9% and 23.4%, respectively. Although SIPL alone gives a modest 9.45% geometric-mean reduction in the incremental study, removing it from the complete method still causes a 23.4% slowdown, showing that state-dependent power retrieval remains important after other overheads are reduced.

We further quantify the trade-off between kernel fusion, register usage, occupancy, and runtime in TABLE 5. The fused kernel uses 89 registers per thread, which limits the theoretical occupancy to 41.67% and results in 37.67% achieved occupancy. Limiting register usage increases achieved occupancy to 66.53% and 89.74%, but it also increases normalized kernel time to 1.60 $\times$ and 2.09 $\times$, respectively. The no-fusion variant reduces register pressure and increases occupancy for the separated kernels, but its normalized runtime remains 1.52 $\times$ due to the redundant gate-state reconstruction and glitch-detection bookkeeping discussed in Section 3.3.

6.4.2 Event-Density-Aware Per-Gate Partitioning vs. Globally Fixed Partitioning. We compare the execution time of the fused power computation kernel under two cycle-partitioning schemes: the event-density-aware per-gate partitioning and the globally fixed partitioning. The event-density-aware per-gate partitioning sets the target event count per thread as 8 ($n_{evt,tgt} = 8$), while the globally fixed partitioning adopts an exponentially stepped parameter sweep (global cycle count per thread n_{cyc}^{fix} ranges from 2 to 32). As shown in Fig. 19, the event-density-aware per-gate partitioning consistently outperforms the globally fixed partitioning, demonstrating the benefits of fine-grained, density-aware parallelism allocation.

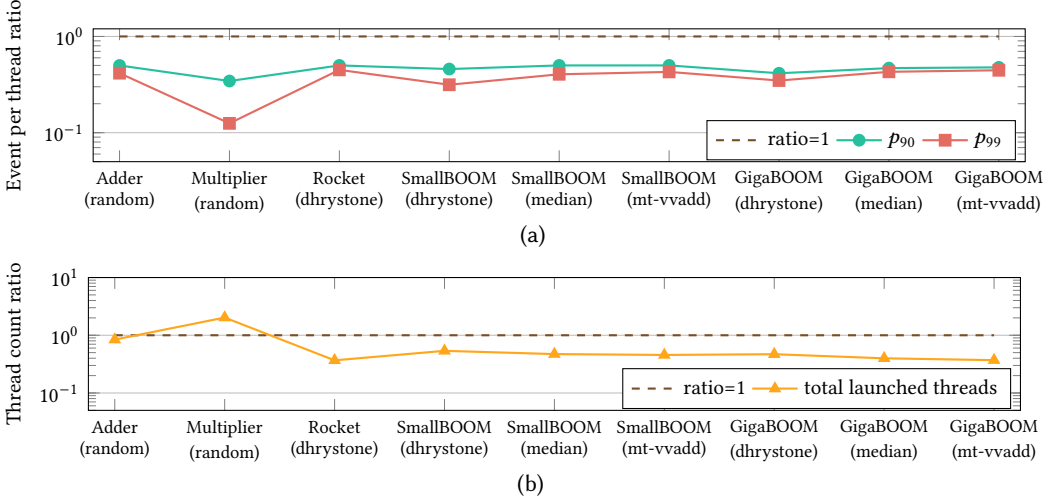


Fig. 20 Ratios (event-density-aware / global-fixed) of (a) events per thread at the tail (p_{90} and p_{99}) and (b) launched thread counts.

Pin threshold	Adder (random)	Multiplier (random)	Rocket (dhrystone)	SmallBOOM (dhrystone)	SmallBOOM (median)	SmallBOOM (mt-vvadd)	GigaBOOM (dhrystone)	GigaBOOM (median)	GigaBOOM (mt-vvadd)
4	0%	11.05%	35.268%	34.976%	34.976%	34.976%	51.779%	51.779%	51.779%
8	0%	0%	0.044%	0.084%	0.084%	0.084%	0.052%	0.052%	0.052%
16	0%	0%	0.044%	0.084%	0.084%	0.084%	0.052%	0.052%	0.052%
32	0%	0%	0.017%	0.04%	0.04%	0.04%	0.027%	0.027%	0.027%

Fig. 21 Heatmap of fallback-gate ratios for different SIPL pin-count thresholds across benchmarks.

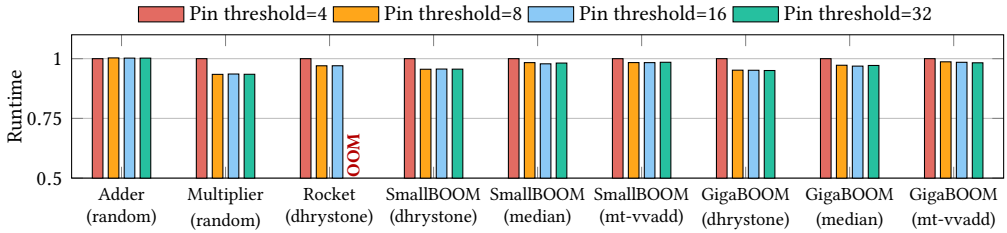


Fig. 22 Fused kernel runtime with different SIPL pin thresholds. Normalized to pin threshold=4. OOM denotes out of memory.

To investigate the underlying reasons, we compare the thread statistics of event-density-aware partitioning ($n_{\text{evt,tgt}} = 8$) and the best globally fixed setting ($n_{\text{cyc}}^{\text{fix}} = 8$). Fig. 20 reports, for each dataset, the ratios (event-density-aware / global-fixed) of tail events per thread and thread counts. As shown in Fig. 20a, the event-density-aware scheme lowers events per thread at the tail (p_{90} and p_{99}) compared with the globally fixed scheme. This indicates that the longest per-thread processing chains are shortened, i.e., the effective thread depth is reduced. Combined with the thread count reduction in Fig. 20b, these results suggest that the event-density-aware scheme redistributes parallelism toward dense gates and compresses tail depth without increasing the total thread count.

6.5 Parameter Sensitivity Analysis

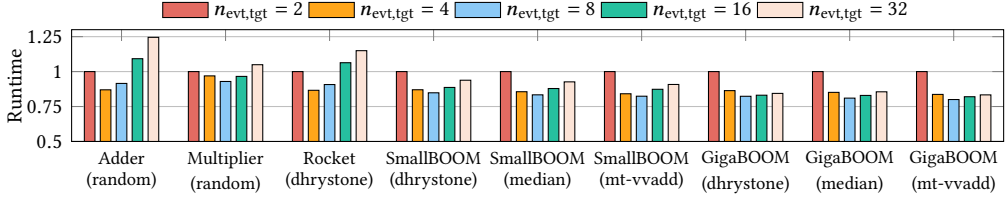


Fig. 23 Fused kernel runtime with different target event count per thread $n_{\text{evt,tgt}}$. Normalized to $n_{\text{evt,tgt}} = 2$.

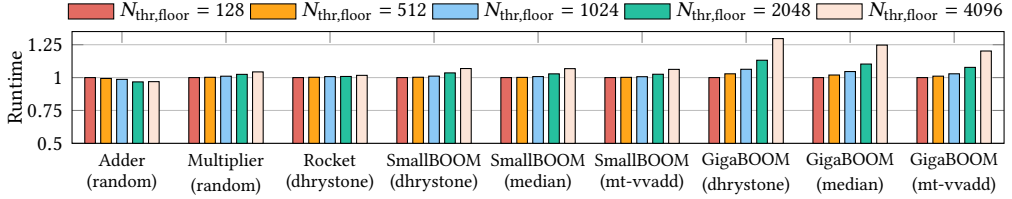


Fig. 24 Fused kernel runtime with different parallelism floor $N_{\text{thr,floor}}$. Normalized to $N_{\text{thr,floor}} = 128$.

6.5.1 SIPL Fallback Threshold. We tune the SIPL fallback threshold, which controls the maximum pin count handled by direct SIPL indexing. As shown in Fig. 21, threshold 4 yields a high fallback-gate ratio (34.1% on average), while thresholds 8 and 16 reduce it to 0.05% on average. Fig. 22 shows that thresholds 8 and 16 reduce geometric-mean runtime by 2.9% over threshold 4, whereas threshold 32 gives no meaningful gain and causes out-of-memory on one benchmark. We therefore choose threshold 16 to keep fallback traversal rare while avoiding the memory risk.

6.5.2 Event-Density-Aware Partitioning Parameters. We analyze event-density-aware partitioning parameters and use the observed trends to explain the selected defaults. For $n_{\text{evt,tgt}}$, we fix $N_{\text{thr,floor}} = 512$ and sweep values from 2 to 32 with exponential steps. Intuitively, a smaller $n_{\text{evt,tgt}}$ (finer granularity) would improve parallelism and thus reduce kernel execution time. However, as shown in Fig. 23, experiments show a counterintuitive trend: an excessively small $n_{\text{evt,tgt}}$ leads to a higher kernel execution time. This is because, as given in Equation (6), the overhead of gate-state reconstruction scales with the number of threads. A smaller $n_{\text{evt,tgt}}$ results in more threads and therefore incurs higher reconstruction overhead. This illustrates a fundamental trade-off between the degree of parallelism and the resulting overhead. Our results indicate that $n_{\text{evt,tgt}} = 8$ achieves the optimal balance, minimizing runtime while maintaining sufficient parallelism.

We also analyze the parallelism floor $N_{\text{thr,floor}}$ with $n_{\text{evt,tgt}} = 8$, as shown in Fig. 24. Compared with $N_{\text{thr,floor}} = 128$, $N_{\text{thr,floor}} = 512$ increases geometric-mean runtime by only 0.7%, while larger floors of 1024, 2048, and 4096 increase it by 1.8%, 4.4%, and 10.3%, respectively. This matches the reconstruction-overhead trade-off: larger floors launch more threads for sparse gates. We therefore choose $N_{\text{thr,floor}} = 512$ because it is close to the best runtime with a higher minimum floor.

7 Related Work

Relevant prior work spans academic power analysis research, practical tools, and GPU acceleration for electronic design automation (EDA). We first review academic power-analysis research, which can be broadly classified into ML-based and simulation-based approaches.

In the context of ML-based methods, gate-level power analysis research primarily employs RTL switching activity data to train predictive models for estimating power characteristics at the gate level. PRIMAL [30] introduces an ML framework that encodes RTL register switching activities into learnable feature embeddings, thereby predicting gate-level cycle-accurate power waveform from RTL simulation traces. GRANNITE [28] estimates average power with a graph convolutional

network (GCN) that propagates toggle rates through combinational logic, and GRIPT [19] improves it with feature-aware attention for untrained-cell prediction across technology nodes. More recently, ATLAS [12] and a GNN method [14] predict cross-design, cycle-level power from gate-level netlists. Regarding the RTL level, MasterRTL [6] proposes a framework using operator graph representations to facilitate pre-synthesis average power estimation. APOLLO [26] presents an automated RTL power modeling framework that enables cycle-accurate power estimation with proxy selection and linear regression. Overall, ML-based approaches typically *estimate* power from learned design representations and thus neither guarantee *per-gate*, *per-event* time-based correctness nor generate a *complete* report covering all power components, peak power, and waveforms.

In the domain of simulation-based methods, Strober [10] accelerates power estimation by sparsely sampling RTL snapshots through field-programmable gate array (FPGA) emulation and replaying them using commercial power analysis tools. SMART-GPO [17] employs a randomized sampling approach that samples only 1% of cycles for efficient glitch power estimation. These simulation-based approaches are more physically grounded than purely ML-based estimation, but they still introduce *approximations* through sampling and limited temporal coverage.

Practical tools also cover related flows: OpenSTA [1] provides static timing analysis and supports *averaged* power analysis, but it does not perform the gate-level time-based power computation targeted in this work. Trace2Power [2] *exports switching-activity* for downstream power-analysis flows rather than computing a complete power report. Commercial signoff tools such as Synopsys PrimeTime PX [21] support gate-level time-based power analysis and therefore provide the closest baseline for validating complete power-computation results.

Finally, GPU acceleration has been widely explored for various EDA tasks, such as logic simulation [5, 27, 29, 31], placement [15], routing [7, 13], and STA [8, 9]. Notably, GATSPI [29] accelerates the delay-aware logic re-simulation step in a gate-level power-improvement flow and generates switching activity for downstream power analysis. It addresses *activity generation* rather than the gate-level time-based power analysis stage accelerated in this paper. More recently, HeteroPower [20] proposes a CPU-GPU heterogeneous engine for gate-level averaged power analysis that reuses GPU-resident STA data to accelerate power computation. However, it targets *averaged* power analysis rather than event-driven time-based power analysis.

Overall, prior work either estimates or samples power behavior, provides averaged-power analysis or activity-export functionality, or supports complete gate-level time-based power analysis in CPU-based commercial flows, leaving this complete analysis flow underexplored on GPUs.

8 Conclusions

In this paper, we propose, to the best of our knowledge, the first GPU-accelerated gate-level time-based power analysis framework. We introduce a state-indexed power lookup method to address inefficient state-dependent power retrieval. We further propose an event-density-aware, per-gate cycle partitioning strategy that exploits variations in gate event density to allocate an appropriate number of GPU threads to each gate. Finally, we present a kernel fusion strategy that computes dynamic and leakage power within a single kernel, thereby reducing redundant work in separate kernels. Our framework achieves high accuracy and a 1.94× to 37.63× end-to-end speedup over multi-threaded PrimeTime PX.

References

- [1] [n. d.]. OpenSTA. <https://github.com/The-OpenROAD-Project/OpenSTA>. [Accessed 29-10-2024].
- [2] [n. d.]. Trace2power. <https://github.com/antmicro/trace2power>. [Accessed 12-5-2026].
- [3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao,

- Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 10–21.
- [4] Jayaram Bhasker and Rakesh Chadha. 2009. *Static timing analysis for nanometer designs: A practical approach*. Springer Science & Business Media.
- [5] Debapriya Chatterjee, Andrew Deorio, and Valeria Bertacco. 2011. Gate-Level Simulation with GPU Computing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16, 3, Article 30 (2011), 26 pages.
- [6] Wenji Fang, Yao Lu, Shang Liu, Qijun Zhang, Ceyu Xu, Lisa Wu Wills, Hongce Zhang, and Zhiyao Xie. 2023. MasterRTL: A Pre-Synthesis PPA Estimation Framework for Any RTL Design. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
- [7] Zizheng Guo, Feng Gu, and Yibo Lin. 2022. GPU-Accelerated Rectilinear Steiner Tree Generation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
- [8] Zizheng Guo, Tsung-Wei Huang, Zhou Jin, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*. 1–6.
- [9] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
- [10] Donggyu Kim, Adam Izraelevitz, Christopher Celio, Hokeun Kim, Brian Zimmer, Yunsup Lee, Jonathan Bachrach, and Krste Asanović. 2016. Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*. 128–139.
- [11] William R Knight. 1966. A computer method for calculating Kendall's tau with ungrouped data. *J. Amer. Statist. Assoc.* 61, 314 (1966), 436–439.
- [12] Wenkai Li, Yao Lu, Wenji Fang, Yugao Zhu, Ziyang Guo, Jing Wang, Mengming Li, Qijun Zhang, and Zhiyao Xie. 2026. A Self-Supervised and Cross-Design Netlist Power Model for Time-Based Layout Power Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2026).
- [13] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2023. GAMER: GPU-Accelerated Maze Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 42, 2 (2023), 583–593.
- [14] Shih-Chun Lin, Yung-Chih Chen, and Bo-Hao Huang. 2026. Enabling Cross-Design Power Trace Prediction with GNNs for Gate-Level Netlists. In *IEEE/ACM Design, Automation and Test in Europe (DATE)*. 1–7.
- [15] Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Bruce Khailany, and David Z. Pan. 2021. DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40, 4 (2021), 748–761.
- [16] Janet Olson, Ivailo Nedelchev, Yuegin Danny Lin, Ashutosh S Mauskar, and James Sproch. 1998. State dependent power modeling. US Patent 5,838,579.
- [17] Yikang Ouyang, Yuchao Wu, Dongsheng Zuo, Subhendu Roy, Tinghuan Chen, Zhiyao Xie, and Yuzhe Ma. 2025. SMART-GPO: Gate-Level Sensitivity Measurement with Accurate Estimation for Glitch Power Optimization. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 1392–1398.
- [18] Jakob Progsch and Václav Zeman. [n. d.]. ThreadPool: A Simple C++11 Thread Pool Implementation. <https://github.com/progschj/ThreadPool>. [Accessed 10-6-2026].
- [19] M. B. Rakesh, Pabitra Das, K. R. Sai Pranav, and Amit Acharyya. 2025. GRIPT: Graph Attention-Assisted Inductive Methodology for Fast and Accurate Average Power Estimation From RTL Simulation Skipping Gate-Level Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 44, 11 (2025), 4209–4221.
- [20] Xizhe Shi, Zizheng Guo, Zuodong Zhang, Yun Liang, Runsheng Wang, and Yibo Lin. 2026. HeteroPower: A CPU-GPU Heterogeneous Engine to Accelerate Gate-Level Power Analysis. In *Proceedings of the IEEE/ACM International Symposium of EDA (ISEDA)*.
- [21] Synopsys. [n. d.]. PrimeTime: Static Timing Analysis. <https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html>. [Accessed 25-5-2026].
- [22] Synopsys. [n. d.]. Verdi. <https://www.synopsys.com/verification/debug/verdi.html>. [Accessed 19-7-2025].
- [23] Synopsys. [n. d.]. What is glitch power? <https://www.synopsys.com/glossary/what-is-glitch-power.html>. [Accessed 11-7-2025].
- [24] Synopsys. 2023. *PrimePower and PrimePower RTL User Guide*.
- [25] Neil HE Weste and David Harris. 2015. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India.
- [26] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das. 2021. APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.
- [27] Cheng Zeng, Fan Yang, and Xuan Zeng. 2021. Accelerate Logic Re-simulation on GPU via Gate/Event Parallelism and State Compression. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

- [28] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2020. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
- [29] Yanqing Zhang, Haoxing Ren, Akshay Sridharan, and Brucek Khailany. 2022. GATSPI: GPU accelerated gate-level simulation for power improvement. In *ACM/IEEE Design Automation Conference (DAC)*. 1231–1236.
- [30] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. 2019. PRIMAL: Power Inference using Machine Learning. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
- [31] Yuhao Zhu, Bo Wang, and Yangdong Deng. 2011. Massively Parallel Logic Simulation with GPUs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16, 3, Article 29 (2011), 20 pages.