Late Breaking Results: Statistical Timing Graph Scheduling Algorithm for GPU Computation

Chih-Chun Chang[1] and Tsung-Wei Huang[1]
[1]*University of Wisconsin–Madison, USA*
{chih-chih.chang, tsung-wei.huang}@wisc.edu

***Abstract*—Statistical Static Timing Analysis (SSTA) is a crucial technique in digital circuit design because it addresses on-chip variations (OCV) by propagating timing distributions instead of fixed delays. However, the computational complexity of SSTA demands significant memory and long runtimes. While GPUs offer opportunities to accelerate SSTA, their limited memory capacity makes it challenging to handle large-scale SSTA workloads. To address this challenge, we propose a statistical timing graph (STG) scheduling algorithm combined with a GPU memory management strategy. We have shown up to 4.9× speedup on a GPU with 16 GB memory compared to a 20-thread CPU baseline when solving an 18.2 GB STG.**

I. Introduction

Statistical static timing analysis (SSTA) enhances digital circuit design by providing more accurate delay estimates than traditional STA [1]. By modeling delays as probability distributions, SSTA effectively accommodates various on-chip variation (OCV). For instance, [2] uses Monte Carlo simulation to evaluate timing yield as the fraction of samples meeting timing constraints. However, the method is computationally intensive due to the extensive sample exploration required. To address this, [3] implements a GPU-based Monte Carlo SSTA engine, achieving speedups of up to two orders of magnitude compared to a CPU-based implementation. Despite this, performing incremental analysis using Monte Carlo-based SSTA remains a significant challenge because of the re-sampling overhead. In contrast, the block-based SSTA methods [4] represent gate delays and arrival times as random variables and propagate these quantities through the timing graph using statistical operations as shown in Figure 1. Gate delays are represented as probability density functions (PDFs), with multiple PDFs assigned to handle various corner cases. Arrival times are represented as cumulative distribution functions (CDFs), which propagate through the circuit. Due to this linearity, block-based SSTA is more efficient than sampling-based counterparts.

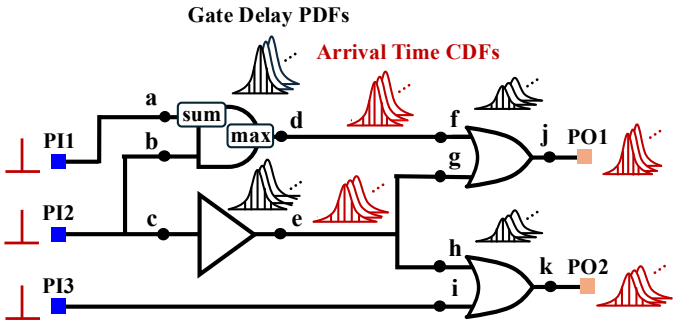

Figure 1: Illustration of a block-based SSTA problem. Gate delays and arrival times are modeled as random variables and propagated through the circuit graph using statistical max and sum operators.

With increasing circuit sizes and shrinking transistor dimensions, more OCVs emerge due to manufacturing process variations. For example, completing block-based SSTA on large designs with over three million pins [5], while accounting for tens of early and late arrival times per pin and more than a dozen corner files can require memory exceeding hundreds of gigabytes (GB). Such significant memory demand poses a major challenge for accelerating block-based SSTA using GPUs which typically have limited memory capacities compared to CPU (e.g., 24 GB memory for NVIDIA GeForce RTX 4090). To overcome the challenge, we introduce an STG scheduling algorithm combined with a GPU memory management strategy to improve the SSTA runtime performance on GPU. We've evaluated the performance using timing graphs generated by [6]. Compared to a 20-thread CPU baseline, our method achieves a $4.9\times$ speedup when solving an 18.2 GB STG. We plan to open-source our algorithms to benefit the timing research community.

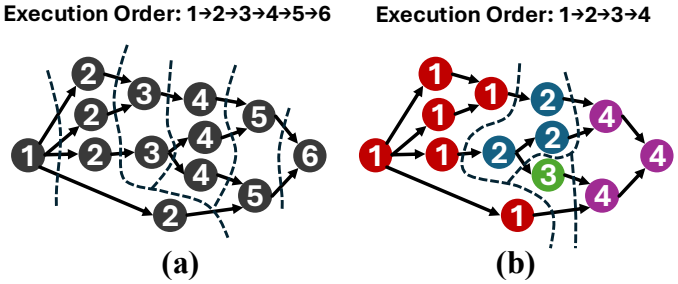

Figure 2: (a) The STG of the circuit, shown in Figure 1, is structured with the execution order assigned to each node based on the topological sorting algorithm. (b) Our scheduling algorithm generates a scheduled STG with an execution order assigned to each node for GPU processing.

II. Problem Definition and Challenges

Given a statistical timing graph (STG), where nodes represent pins with timing quantities and edges represent dependencies, and a target GPU memory capacity, our goal is to schedule the STG with an optimized execution order that maximizes GPU memory utilization, minimizes CPU-GPU communication, and enhances GPU parallelism to improve SSTA runtime. We focus on circuits that fit within the CPU's main memory but may exceed the GPU's memory capacity as the amount of timing data increases. This formulation arises from our collaboration with industrial partners, highlighting a common challenge in accelerating large SSTA workloads using GPUs.

Scheduling STG on a GPU is challenging due to limited memory, requiring frequent data swapping and strict execution order to ensure correctness. A node can only execute after its parent nodes have completed their computations, as seen in Figure 2(b), where tasks in Level 2 must wait for Level 1 outputs. Additionally, maximizing GPU parallelism is crucial for accelerating SSTA runtime, requiring efficient resource utilization while respecting data dependencies. Prioritizing data transfers that enhance GPU occupancy is key to

achieving optimal performance. Furthermore, effective memory management is essential, as limited GPU memory necessitates careful tracking and timely swapping of data to prevent out-of-memory issues.

III. Algorithm

To address the above challenges, we propose a STG scheduling algorithm and a GPU memory management strategy designed for efficiently solving SSTA problems on GPUs.

The scheduling algorithm manages STG dependencies to ensure correct execution order while optimizing data transfers to maximize GPU parallelism. Each node in the STG has two dependency counters: *predecessor* and *successor*. The *predecessor* counter ensures that all parent nodes of a node are present in GPU memory before it can begin to schedule. When a node's timing data is transferred to GPU, the *predecessor* counters of its child nodes are incremented. Once the counter value of a node matches its in-degree, the node is ready to schedule. Conversely, the *successor* counter, initialized to the node's out-degree, tracks the number of unfinished child nodes. Upon completing its computation, a node decrements the *successor* counters of its parent nodes. If a *successor* counter reaches zero, it indicates that the node's data is no longer needed and can be evicted from GPU memory, freeing space for other computations.

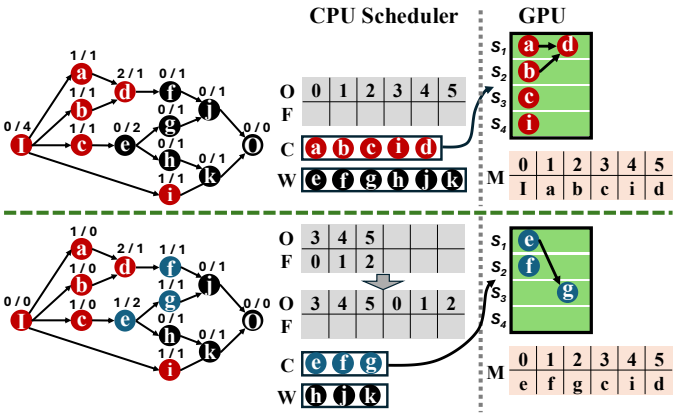

Figure 3: Illustration of our scheduling algorithm. Each node in the STGs on the left is equipped with a *predecessor* and *successor* counter, which are displayed at the top of each node. The upper part illustrates how the first execution order node in Figure 2 is scheduled by our scheduler and how its data is transferred to GPU memory. The lower part shows the second-order node. **O** represents the *occupied* queue, **F** the *free* queue, **C** the *computation* queue, **W** the *waiting* queue, **M** the GPU memory (with a size of six units), and $\mathbf{S}_i$ the CUDA stream.

Our GPU memory management algorithm integrates with the scheduler to efficiently handle GPU memory limitations. A large memory pool is allocated, managed by two queues: *free* queue (available memory segments) and *occupied* queue (used memory segments). Initially, all memory pointers are in the *free* queue, and the *occupied* queue is empty. Each node's *predecessor* counter starts at zero, and its *successor* counter is initialized to its out-degree. When the scheduler requests memory, a pointer is moved from the *free* queue to the *occupied* queue. After GPU kernels complete, nodes in the *occupied* queue with *successor* counters at zero are evicted, their data copied back to CPU, and their memory pointers returned to the *free* queue. If no memory is available in the *free* queue, the oldest data in the *occupied* queue is evicted, copied to CPU, and its pointer returned to the *free*. Unfinished nodes are rescheduled, and their children's *predecessor* counters are updated. This algorithm ensures efficient memory use and prevents out-of-memory issues (see Figure 3).

IV. Experimental Results

We evaluated the performance of our algorithm on 9 timing graphs derived from industrial circuits provided by [6]. Timing quantities were sampled from a standard normal distribution, with varying sizes assigned to each circuit pin to generate STGs of different scales. We implemented a CPU-based timer in C++ (*cpp*) and a GPU-based timer using cudaFlow [7] (*cuf*) as our baselines. *cpp* constructs a task graph for statistical sum and max operations on STG nodes and utilizes 20 CPU threads, and *cuf* builds the dependencies using CUDA Graph on GPU. All experiments were conducted on a 64-bit machine equipped with an Intel processor (20 threads, 128 GB RAM) and an NVIDIA A4000 GPU (16 GB memory). The interconnect bandwidth between the CPU and GPU is 25 GB/s bidirectional.

Circuit	‖V‖ / ‖E‖	*Size*	*cpp*	*cuf*	*ours*
aes_core_1		1.0	679	15405	370 (1.8×)
aes_core_2	66K / 86K	2.0	1306	15599	450 (2.9×)
aes_core_3		3.1	1870	15576	396 (4.7×)
des_perf_1		4.6	3106	OOM	1813 (1.7×)
des_perf_2	303K / 387K	9.2	5798	OOM	2108 (2.7×)
des_perf_3		13.8	8474	OOM	1760 (4.8×)
vga_lcd_1		6.0	4051	OOM	2367 (1.7×)
vga_lcd_2	397K / 489K	12.1	7651	OOM	2271 (3.4×)
vga_lcd_3		18.2	11214	OOM	2297 (4.9×)

Table I: Runtime performance comparison (milliseconds) among *cpp*, *cuf* and *ours* across various STG sizes (GB). OOM denotes out-of-memory on the GPU.

Table I presents a runtime performance comparison among *cpp*, *cuf* and our algorithm across various STG sizes. Our algorithm consistently outperforms *cpp* and *cuf* in all benchmarks. For instance, the *aes_core_3* circuit (3.1 GB) achieves a 4.7× speedup over *cpp*. Similarly, the *des_perf_3* circuit (13.8 GB) and *vga_lcd_3* circuit (18.2 GB) achieve speedups of 4.8× and 4.9×, respectively. As shown in the table, the performance improvement becomes more significant with larger circuit sizes, demonstrating the scalability of our algorithm. Additionally, *cuf* encounters GPU out-of-memory (OOM) issues on larger circuits due to the GPU memory limitation. In contrast, our scheduling algorithm with the GPU memory management strategy overcomes this limitation.

To further contribute to the timing analysis research community, we plan to open-source our algorithms.

References

[1] 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.

[2] R. Kanj, R. Joshi, and S. Nassif, "Mixture importance sampling and its application to the analysis of sram designs in the presence of rare failure events," in *ACM/IEEE DAC*, 2006, pp. 69–72.

[3] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *IEEE/ACM ASPDAC*, 2009.

[4] A. Devgan and C. Kashyap, "Block-based static timing analysis with uncertainty," in *IEEE/ACM ICCAD*, 2003, pp. 607–614.

[5] J. Hu, G. Schaeffer, and V. Garg, "Tau contest on incremental timing analysis," in *ICCAD*. Austin, TX, USA: IEEE, 2015.

[6] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. Wong, "Opentimer v2: A new parallel incremental timing analysis engine," *TCAD*, 2020.

[7] D.-L. Lin and T.-W. Huang, "Efficient gpu computation using task graph parallelism," in *Euro-Par*, Springer. Cham, Switzerland: Springer, 2021.