Analyzing Timing in Shorter Time: A Journey through Heterogeneous Parallelism for Static Timing Analysis

Zizheng Guo[1,2], Yibo Lin[1,2,3]* Runsheng Wang[1,2,3], Ru Huang[1,2,3]
[1]School of Integrated Circuits, Peking University [2]Institute of Electronic Design Automation, Peking University
[3]Beijing Advanced Innovation Center for Integrated Circuits

***Abstract*—This paper reviews recent work on the acceleration of static timing analysis (STA), with a special focus on parallel and heterogeneous computing techniques. Timing analysis is one of the most critical tasks in circuit design. The ever-increasing size and complexity of modern circuit design has asked for unprecedented STA runtime speed-up which has to be achieved through CPU-GPU heterogeneous computing. GPU-accelerated STA is however difficult due to its nature of irregular computation and memory access patterns. We demonstrate and analyze the algorithm design and scheduling considerations in recent works targeting various different STA stages, and discuss future directions of STA acceleration as well as the future of timing optimization in heterogeneous circuit design flow.**

***Index Terms*—Static timing analysis, Heterogeneous computing**

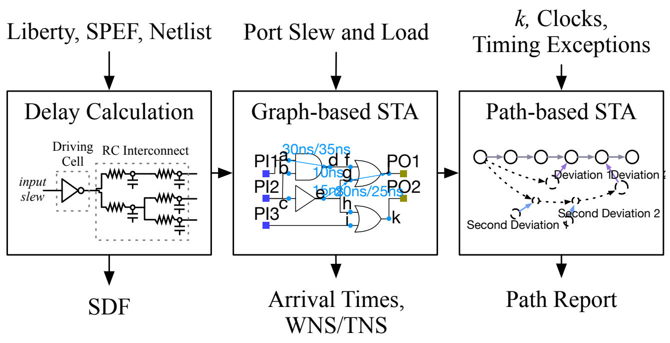

Fig. 1: Overview of STA steps, inputs, and outputs.

I. Introduction

The analysis and optimization of timing is one of the most critical tasks in circuit design as it is directly tied to the chip's correctness and runtime performance. With the ever-increasing size and complexity of modern circuit design as well as the ever-rising difficulty to model physics in advanced nodes, the runtime it takes to complete a design cycle has become much longer. This runtime burden is worsened by the increasing demand to explore and search the design space more thoroughly for power, performance, and area (PPA) improvements. Within this runtime burden, static timing analysis (STA) has become a remarkable bottleneck.

During the circuit design flow, STA will be invoked for hundreds to thousands of times because most design steps like synthesis, placement, and routing are iterative. As a result, the performance of STA is critical for efficient design turnaround. STA consists of graph algorithms running on large circuit graphs with millions of nodes and edges. An efficient STA engine thus calls for massive parallelism that is beyond the reach of traditional multi-core CPUs.

In this paper, we survey recent works on GPU-accelerated STA. These works focus on different STA stages including graph-based analysis, path-based analysis, delay modeling, etc. We review various techniques on GPU-friendly algoithms, data structures, scheduling algorithms, and other engineering efforts to overcome the inherent difficulty in STA parallelization such as irregular graph computation workload and memory access patterns.

The rest of the paper is organized as follows. Section II introduces the background of STA in a heterogeneous computing perspective. Then, Sections III, IV, and V reviews recent work categorized into different STA stages including delay modeling, graph-based analysis, and path-based analysis respectively. Finally, Section VI concludes the review and discuss future directions.

II. Preliminaries

STA takes as input a circuit represented as a directed acyclic graph (DAG) with nodes indicating pins and edges indicating signal connections. Along with the circuit, STA also takes physical information such as cell library in process design kit (PDK) as well as parasitics information for circuit interconnects. With these inputs and further clock and path settings, STA analyzes design timing and outputs worst negative slack (WNS), total negative slack (TNS), and a set of critical logic paths [1].

STA is usually divided into 3 steps, as shown in Figure 1:

*Corresponding author: Yibo Lin (yibolin@pku.edu.cn).

1) Delay calculation. This step makes use of parasitics information to derive a compact modeling of the metal interconnects.
2) Graph-based analysis. This step propagates voltage slew and signal arrival time of pins throughout the circuit graph in topological order. After this step, WNS and TNS as well as pin slacks can be derived.
3) Path-based analysis. This step searches for top-k most critical signal paths with k given by designers. Optionally, the path slacks are recalculated based on path-specific conditions for better accuracy.

The performance of STA is critical as it is frequently used in circuit design. The runtime of delay calculation and graph-based STA is proportional to the scale of circuit DAG, which can exceed millions or even more pins and arcs. The runtime of path-based STA is further multipled by the number of paths k requested, which may range from tens to thousands. A typical STA run on a million-sized design can take tens of minutes to hours and become a bottleneck in the physical design flow.

Parallel computation is the key to accelerating STA due to its problem scale. Current major STA engines like OpenSTA [2], PrimeTime [3], and OpenTimer [4] all supports parallel STA using multi-core CPUs. It has been widely observed, however, that CPU-based parallel STA cannot scale beyond 8–16 CPU threads [4], and the performance may even degrade after that threshold.

Heterogeneous computing using general-purpose graphics computing units (GPGPU) has been shown to provide unprecedented speed-up on a variety of tasks, with machine learning a notable example. However, successful heterogeneous application requires balanced and regular patterns in computation and memory access, which is hardly the case for STA. Delay calculation requires solving interconnect equations in a long-tailed net size distribution, which incurs workload imbalance between working threads. Graph-based and path-based analyses work on highly irregular circuit graph with induced task dependencies and irregular memory access. These make GPU-accelerated STA quite challenging.

III. Delay Modeling

Delay modeling or delay calculation is the first step in STA and it determines the analysis accuracy when choosing from different delay models. This section introduces GPU-accelerated delay calculation works arranged inside a brief review of different delay models. One can refer to [5], [6] for a more comprehensive introduction of delay models.

There are usually two separate delay models inside a STA engine for net and cell delay calculation, respectively. Net delay model analyzes the voltage response of metal interconnects represented in resistors and capacitors (RC) networks. Cell delay model characterizes the standard cell voltage response into compact forms like look-up tables conditioned on the environment of the cell. Voltage response consists of signal delay and transition time.

A. Heterogeneous net delay model

One widely used and simple net delay model is the Elmore delay model [7]. Elmore delay approximates net delay and slew by summing up RC products along the tree path from the source to every sink. In reality, this process is implemented as a tree-based dynamic programming algorithm [4]. Guo Z. et al [8], [9] present a GPU-accelerated Elmore delay calculator on top of OpenTimer. As GPU has very limited call stack, they propose a sweeping-based algorithm to simulate the dynamic programming on GPU. An Elmore delay speed-up of 2.54$\times$ and overall speed-up of 3.69$\times$ over OpenTimer has been reported.

As process technology continues to develop, high-order voltage effects including resistive shielding have evolved in sub-14nm nodes. This calls for more accurate interconnect modeling than the Elmore approximation. To this end, Arnoldi models and other model order reduction techniques have been proposed [10], [11], [12]. They construct the linear system equation of each RC interconnect, and then reduce the system order using algebraic tricks. Guo Z. et al [13] present a GPU-accelerated Arnoldi delay calculator based on the coordinate-transformed Arnoldi algorithm [12] and reported 7.27$\times$ speed-up over PrimeTime. Advanced net delay models face more severe workload imbalance challenge because both the interconnect size variation and the time complexity of interconnect modeling are higher. They tackled this problem by splitting the algebraic computations (e.g., sparse LU decomposition) into multiple stages with different parallelism to exploit.

B. Heterogeneous cell delay model

The most widely used cell model is called non-linear delay model (NLDM) which is often combined with the Elmore model to derive delays for both nets and cells. NLDM models cell timing arcs as linearly-interpolated look-up tables (LUTs) indexed by input voltage transition and output capacitive load. A look-up table query needs to access two index arrays to locate interpolation point, and then access the result matrix 4 times. The CASTA timer presented by Wang H. et al [14] propose to optimize such memory access by placing indices and values close to each other (table-index remapping) and inside texture memory. They reported up to 14.89$\times$ speed-up on NLDM calculation.

As transistors shrink to nanoscale, current-source models (CSM) begin to replace NLDM for its better accuracy especially in signoff scenarios. CSM does not directly model the delay and transition of cell arcs, but instead models a cell as a time- and voltage-controlled current source (for composite current source model, CCS [15], [16]) or voltage source (for effective current source model, ECSM [17]). Lin S. et al [18] propose a GPU-accelerated CCS model calculator. They propose an efficient matrix inverse precomputation technique making use of the similarity of conductance matrices and achieve up to 3.4× speed-up in 2% error.

IV. Graph-Based Analysis

A delay calculator is itself not an end-to-end STA engine without graph-based analysis. End-to-end STA engines are more difficult to accelerate on GPU due to the synchronization and data transfer overhead between CPU and GPU. Prior works like [14], [19] can achieve high speed-up ratios when measuring only the kernel runtime. However, their end-to-end performance may even be 0.9× inferior to a CPU flow. As a result, a GPU-accelerated graph-based STA engine must incorporate efficient task-scheduling strategies to overcome the overhead. Guo Z. et al [8] overlaps memory transfer and independent computations using CUDA streams. They make choices on CPU/GPU task placement based on a runtime breakdown to avoid over-optimization. In their journal extension [9], they extend this framework to multi-corner graph-based STA by placing the corner-level parallelism at the GPU thread-level to achieve better scalability up to 25.67×.

Arrival time propagation in graph-based STA is another challenge due to its dependency constraints. To sort out the dependency, levelization is widely used [8], [14] as a preprocessing step to make sure nodes within each topological level can run in parallel. However, levelization itself is shown to take a significant amount of runtime, so GPU-accelerated topological sorting and levelization is designed [8] that accelerates it up to 4.51×. For traditional CPU-based task parallelism, a better partition can also improve the performance by reducing scheduling cost. Zhang B. et al [20] show that such partition can be efficiently generated with the help of GPU.

Besides parallelism within a circuit graph, prior works also explore the application of GPUs or FPGAs in Monte-Carlo-based statistical static timing analysis (SSTA) [21], [22]. SSTA using Monte Carlo provides another parallelism across different independent simulation runs which fits nicely with heterogeneous platforms.

V. Path-Based Analysis

Path searching is the ultimate step in STA flow. The search of top-k critical paths often relies on prefix-suffix tree algorithms [23] or its improvements [24], [25]. The algorithm behaves like a A* search that relies on a priority heap and a first-in-first-out (FIFO) queue. One challenge is that A* algorithms are inherently sequential because only the best current solution (in our formulation, current most-critical path) can be used to expand solution space. Fortunately, it turns out that this A* algorithm constraint can be carefully relaxed and turned into an equivalent iterative-pruning algorithm. Guo G. et al [26], [27] propose the above technique. Their evaluation generates up to 1 million paths with up to 45× shorter runtime than a saturated CPU parallel STA engine. Later, they propose a path-search-oriented graph partitioning strategy [28] to overcome the single-GPU memory limitation.

Path-based analysis is also a stage where various user-defined constraints and exceptions are applied. These constraints, settings, and exceptions often create unique challenges for accelerating STA. One notable example is common path pessimism removal (CPPR)[1]. CPPR requires adjustment of path criticality based on the common clock driving paths between its launching and capturing registers. A brute-force algorithm for CPPR needs to enumerate all pairs of registers which make the path search extremely slow for large designs. Recently, better algorithms called depth-based CPPR have been proposed [29], [30], [31] that clusters registers into groups based on a batch of depth-relevant criticality adjustments. Depth-based CPPR is also implemented on GPU in the HeteroCPPR framework by Guo Z. et al [32]. They make use of the independence between clustered groups to scale their algorithm to multiple GPUs and achieve up to 16× speed-up.

Besides CPPR, other timing exceptions like false paths, multi-cycle paths, path margins, and set delays have been more and more frequently used in modern chip designs. These rules are written in a design constraints file. Each rule gives a path pattern (including from, through, and to) and actions on the paths. For simple inclusive rules in final timing report, Guo G. et al [33] propose constrained subgraph scanning and sub-forest expansion techniques making use of the topological structure. For advanced exclusive rules like false paths, multi-cycle paths, etc., Guo Z. et al propose the HeteroExcept framework [34] that solves most of the common timing exceptions. They prove the NP-hardness of general exclusive rules handling. Practically, their GPU-accelerated exception-aware STA achieves 6.84× speed-up over PrimeTime by various techniques like GPU exception footprinting and copy-on-write algorithms.

VI. Conclusion and Future Challenges

This paper reviews the current state-of-the-art heterogeneous algorithms and frameworks for STA. Heterogeneous CPU/GPU parallelism have introduced runtime benefit to all major STA stages including delay calculation, graph-based analysis, and path-based analysis. While each stage has its unique challenge, the common challenges are irregular computation and memory patterns as well as imbalanced workload. Prior works introduce various algorithm transformation, preprocessing, and scheduling techniques to achieve an end-to-end speed-up ranging from 3× to 45× on different workloads.

[1]also known as clock reconvergence pessimism removal (CRPR).

With these advancements in heterogeneous STA, there are a number of new challenges and opportunities.

- *Efficient inter-GPU partitioning for large circuits.* Modern system on a chip (SoC) design contains more than tens to hundreds of millions of circuit elements. This scale is beyond the capacity of a single GPU node, or even a classical CPU node. Automatic partitioning and hierarchical STA techniques can help in scaling the STA task to multiple GPUs and multiple nodes, which is vital for the successful application of heterogeneous STA.
- *Advanced manufacturing technology.* The development of process node continues to require even more complex circuit delay models, taking into account aging, multi-input switching, electromigration, advanced variation, etc. Specifically, the introduction of 3D ICs create challenge in multi-corner STA due to the explosion of corner count.
- *Timing-driven optimization.* The ultimate goal of timing analysis is to optimize circuit performance based on the evaluation. This not only calls for effective use of fine-grained timing analysis results through heterogeneous timing-driven design algorithms, but also calls for cross-stage prediction of timing results to enable early feedback for both RTL and physical design.

Acknowledgement

This work was supported in part by the Natural Science Foundation of Beijing, China (Grant No. Z230002) and the 111 project (B18001).

References

[1] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, 1st ed. Springer Publishing Company, Incorporated, 2009.
[2] "OpenSTA," https://github.com/The-OpenROAD-Project/OpenSTA.
[3] "Synopsys PrimeTime," http://www.synopsys.com.
[4] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.
[5] J. Croix and D. Wong, "Blade and razor: cell and interconnect delay analysis using current-based models," in *Proc. DAC*, 2003, pp. 386–389.
[6] U. Baur, P. Benner, and L. Feng, "Model order reduction for linear and nonlinear systems: a system-theoretic perspective," *Archives of Computational Methods in Engineering*, vol. 21, no. 4, pp. 331–358, 2014.
[7] W. C. Elmore, "The transient response of damped linear networks with particular regard to wideband amplifiers," *Journal of applied physics*, vol. 19, no. 1, pp. 55–63, 1948.
[8] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.
[9] ——, "Accelerating static timing analysis using cpu-gpu heterogeneous parallelism," *IEEE TCAD*, pp. 1–1, 2023.
[10] A. Odabasioglu, M. Celik, and L. T. Pileggi, "Prima: Passive reduced-order interconnect macromodeling algorithm," *IEEE TCAD*, vol. 17, no. 8, p. 645, 1998.
[11] C. L. Ratzlaff and L. T. Pillage, "Rice: Rapid interconnect circuit evaluation using awe," *IEEE TCAD*, vol. 13, no. 6, pp. 763–776, 1994.
[12] L. Miguel Silveira, M. Kamon, I. Elfadel, and J. White, "A coordinate-transformed Arnoldi algorithm for generating guaranteed stable reduced-order models of RLC circuits," in *Proc. ICCAD*, Nov. 1996, pp. 288–294.
[13] Z. Guo, T.-W. Huang, Z. Jin, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous static timing analysis with advanced delay calculator," in *Proc. DATE*, 2024.
[14] H. H.-W. Wang, L. Y.-Z. Lin, R. H.-M. Huang, and C. H.-P. Wen, "Casta: Cuda-accelerated static timing analysis for VLSI designs," in *Proc. ICPP*. IEEE, 2014, pp. 192–200.
[15] S. Simoglou, I. Lilitsis, N. Blias, and C. Sotiriou, "Full Stage Delay Calculation Using Full Waveform Propagation and Standard Library CCS Model," in *Proc. ISQED*. San Francisco, CA, USA: IEEE, Apr. 2024, pp. 1–8.
[16] D. Garyfallou, S. Simoglou, N. Sketopoulos, C. Antoniadis, C. P. Sotiriou, N. Evmorfopoulos, and G. Stamoulis, "Gate delay estimation with library compatible current source models and effective capacitance," *IEEE TVLSI*, vol. 29, no. 5, pp. 962–972, 2021.
[17] Cadence, "ECSM Library Format." [Online]. Available: https://www.cadence.com/en_US/home/alliances/standards-and-languages/ecsm-library-format.html
[18] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. F. Young, and M. D. Wong, "GCS-Timer: Gpu-accelerated current source model based static timing analysis," in *Proc. DAC*, 2024.
[19] K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *Proc. FPT*. IEEE, 2018, pp. 110–117.
[20] B. Zhang, D.-L. Lin, C. Chang, C.-H. Chiu, B. Wang, W. L. Lee, C.-C. Chang, D. Fang, and T.-W. Huang, "G-PASTA: Gpu-accelerated partitioning algorithm for static timing analysis," in *Proc. DAC*, 2024.
[21] K. Gulati and S. P. Khatri, "Accelerating statistical static timing analysis using graphics processing units," in *Proc. ASPDAC*. IEEE, 2009, pp. 260–265.
[22] J. Cong, K. Gururaj, W. Jiang, B. Liu, K. Minkovich, B. Yuan, and Y. Zou, "Accelerating Monte Carlo based SSTA using FPGA," in *Proc. FPGA*, 2010, pp. 111–114.
[23] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.
[24] K. Zhou, Z. Guo, T.-W. Huang, and Y. Lin, "Efficient critical paths search algorithm using mergeable heap," in *Proc. ASPDAC*, 2022.
[25] C. Chang, T.-W. Huang, D.-L. Lin, G. Guo, and S. Lin, "Ink: Efficient incremental k-critical path generation," in *Proc. DAC*, 2024.
[26] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *Proc. DAC*. ACM, 2021.
[27] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. F. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE TCAD*, pp. 1–1, 2023.
[28] G. Guo, T.-W. Huang, and M. Wong, "Fast sta graph partitioning framework for multi-gpu acceleration," in *Proc. DATE*, 2023, pp. 1–6.
[29] Z. Guo, T.-W. Huang, and Y. Lin, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," in *Proc. DAC*. ACM, 2021.
[30] Z. Guo, M. Yang, T.-W. Huang, and Y. Lin, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," *IEEE TCAD*, pp. 1–1, 2021.
[31] T. Sun and C. Feng, "Dac-cppr: A fast and accurate approach for common path pessimism removal with divide and conquer on the clock tree." IEEE, 2023, pp. 263–268.
[32] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *Proc. ICCAD*. ACM, 2021.
[33] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated critical path generation with path constraints," in *Proc. ICCAD*, 2021, pp. 1–9.
[34] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, "HeteroExcept: A CPU-GPU heterogeneous algorithm to accelerate exception-aware static timing analysis," in *Proc. ICCAD*, 2024.