GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis

Shiju Lin
sjlin@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Guannan Guo
guo.guannan@huawei.com
Huawei Design Automation Lab, HK

Tsung-Wei Huang
tsung-wei.huang@wisc.edu
University of Wisconsin at Madison

Weihua Sheng
sheng.weihua@huawei.com
Huawei Design Automation Lab, HK

Evangeline F.Y. Young
fyyoung@cse.cuhk.edu.hk
The Chinese University of Hong Kong

Martin D.F. Wong
mdfwong@hkbu.edu.hk
Hong Kong Baptist University

ABSTRACT

Composite Current Source (CCS) timing model plays an important role in modern static timing analysis (STA) because it precisely captures the timing behavior of a design at advanced nodes. However, CCS is extremely time-consuming due to its accurate but complicated timing models. To overcome this challenge, we introduce GCS-Timer, a GPU-accelerated CCS-based timing analysis algorithm. Unlike existing methods that perform model order reduction to trade accuracy for speed, GCS-Timer achieves high accuracy through a fast simulation-based analysis using GPU computing. Experimental results show that GCS-Timer can complete CCS analysis with better accuracy and achieve 3.2× faster runtime compared with a 16-threaded industrial standard timer. The source code is available at https://github.com/cuhk-eda/GCS-Timer.

1 INTRODUCTION

Static timing analysis (STA) plays an essential role in the overall design flow because it verifies the expected timing behaviors of a circuit design using dual-mode (min/max) delay analysis. Among various delay models, *non-linear delay model* (NLDM) is one of most commonly used timing models due to its simple driver and receiver models. On the driver side, NLDM characterizes input-to-output delay and output slew as an efficient look-up table (LUT) of input transition time and output load capacitance. On the receiver side, NLDM assumes a single capacitor for the entire transition. As a result, NLDM is extremely efficient in delay calculation and has been widely adopted by existing timers, such as OpenSTA [4], OpenTimer [10, 13], and commercial tools.

Despite the efficiency, NLDM is not sufficient for reflecting the non-linearities of circuits at advanced technology nodes (45nm and beyond). Specifically, NLDM receiver model fails to capture Miller effect, which dominates the accuracy of delay calculation for small-impedance nets. To overcome this problem, *Composite Current Source* (CCS) model characterizes the driver behavior as a time-varying and voltage-dependent current source, which provides accurate response for arbitrary RC networks. CCS also uses two capacitance values for the receiver model to account for the Miller effect. As a consequence, CCS is more accurate at deep submicron yet at the cost of longer runtime because of the complicated computation of the driver and receiver models.

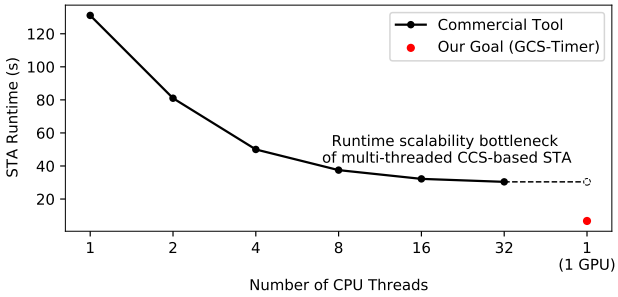

Figure 1: Runtime scalability of CCS-based STA.

To alleviate the long CCS runtime, multi-threading is a common solution. As shown in Figure 1 (reported by an industrial standard timer), a multi-threaded CCS algorithm can reduce the runtime from 120s to 40s when analyzing a 200K-gate design. However, the speedup quickly saturates at 8–16 threads primarily because of the overheads and the limited memory bandwidth of CPU threads [11, 12]. Compared with many-core CPUs, modern GPU supports orders of magnitude more parallelism and much higher memory bandwidth. Because of this advantage, we have seen various GPU-accelerated STA algorithms in recent electronic design automation research [7–9, 14–16]. These results have inspired us to leverage GPU to accelerate CCS analysis.

However, designing a GPU-accelerated CCS algorithm is extremely challenging for three reasons. First, high-accuracy CCS analysis relies on simulation-based methods, which analyze a driver and its receivers together in the RC network. This is very different from NLDM which separates the delay calculation of drivers and RC networks. We need to design a different framework for efficient CCS analysis. Second, CCS delay calculation is very complicated because we need to use the driver model to compute accurate driver response for every simulation iteration, similarly for calculating the capacitance of a receiver. We need to design new efficient GPU kernels that can handle the frequent driver response computation

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK14218422).

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3655983

and the receiver capacitance change during simulation. Third, CCS-based STA analyzes an RC network in many stages for different driver timing arcs and transitions. These analyses share the same RC network and differ only in the driver and receivers, resulting in similar conductance matrices for simulation. Since matrix inverse computation is time-consuming but necessary for simulation, we need new techniques to efficiently compute the inverses of a batch of similar matrices.

To overcome these challenges, we propose GCS-Timer, a GPU-accelerated CCS-based STA algorithm. As shown in Figure 1, our goal is to break the bottleneck of multi-threaded CCS analysis by harnessing power of massively-parallel GPU computing. We summarize three technical contributions below:

- We design a GPU acceleration framework for CCS-based STA. The framework uses a two-pass simulation scheme for high accuracy and fully explores different parallelisms from the two simulation passes.
- We develop an efficient GPU kernel for CCS delay calculation. The kernel has an accurate and lightweight driver current computation engine and an iterative approximation method to break the inter-dependency between the driver response and the circuit while ensuring high accuracy.
- We propose a new precomputation technique for efficient matrix inverse calculation, which is the core of simulation. This technique can identify a group of stages that have similar conductance matrices, and precompute some partial inverse results. The precomputation is efficient, and can significantly reduce the runtime of matrix inverse computation for CCS analysis.

We evaluate the performance of GCS-Timer on a set of open-source circuit benchmarks. Compared with a 16-threaded industrial standard timer, GCS-Timer achieves 3.2× faster runtime and higher accuracy.

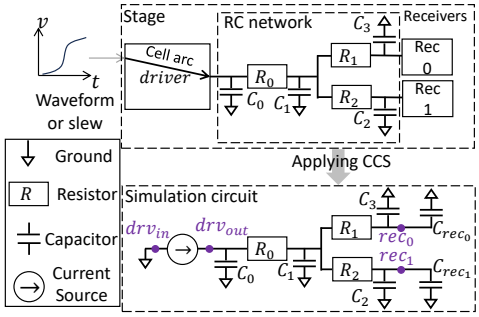

Figure 2: A stage consists of a driver cell arc connected to the receivers by an RC network, which becomes a circuit with current sources, resistors, and capacitors after applying the CCS model.

2 CCS-BASED STAITC TIMING ANALYSIS

Static timing analysis (STA) examines the timing of a design by checking all timing paths. It typically has two procedures, delay calculation and delay propagation, which calculate the delay of each cell/net arc and propagate the delay along each path, respectively.

2.1 Composite Current Source (CCS) Model

The CCS timing model consists of a driver model and a receiver model. The driver model describes how a timing arc propagates a transition from input to output, and how it drives an arbitrary RC network. The receiver model describes the capacitance that an input pin presents to the driver cell.

CCS Driver Model. The CCS driver model is a time- and voltage-dependent current source. It achieves high accuracy by mapping arbitrary transistor behavior for lumped loads to the behavior for a detailed parasitic network. The model is given by a 2D table indexed by input transition time (slew) and output load capacitance. Each table entry describes the driver output current as a piecewise linear function of time, $i = i(t)$.

CCS Receiver Model. CCS models a receiver as a capacitor with varying capacitance depending on input slew and output capacitance. CCS provides two tables for capacitance calculation to account for the Miller effect, which are used before and after the receiver input pin reaching the delay threshold respectively.

CCS delay calculation is typically performed on a stage consisting of a driver cell arc connected to some receiver cells by an RC network. An RC network is a circuit consisting of capacitors to ground and resistors. Figure 2 shows an example stage consisting of two receiver cells and an RC network with 4 capacitors and 3 resistors. This stage becomes a circuit of current sources, resistors and capacitors after applying the CCS model (replacing the driver and receivers by the respective CCS models), as shown in Figure 2.

Given a transition time of the driver input pin, the goal of stage delay calculation is to calculate the response (waveform) at the driver output and receiver input pins. For example, the task in Figure 2 is to compute the waveforms at drv_{out}, rec_0 and rec_1. Delay and slew can be calculated by definition using these waveforms. For example, the delay between drv_{out} and rec_0 is the time difference when they reach a voltage threshold (usually $0.5V_{dd}$).

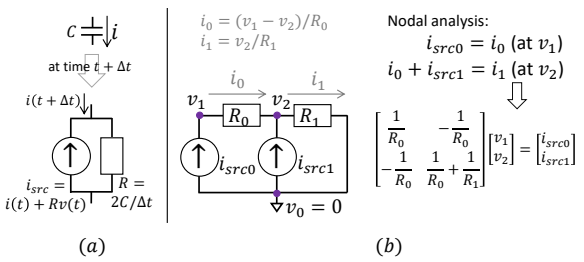

Figure 3: Simulation of circuits with capacitors, current sources and resistors. (a) Transform each capacitor into a current source and a resistor in parallel. (b) Perform nodal analysis to compute the voltage at each node.

2.2 Circuit Simulation

GCS-Timer uses circuit simulation, a slow but accurate method, for stage delay calculation. We leverage a standard approach [17] to simulate a circuit with current sources, resistors, and capacitors.

Simulation starts with an initial state at $t^{(0)}$ and iteratively computes the circuit states at timesteps $t^{(1)}, t^{(2)}, ..., t^{(n)}$. Between consecutive timesteps, We assume a linear current change to simplify

the circuit and solve for the circuit state. For a capacitor C, the current flowing through it is $i = C\frac{dv}{dt}$. By computing the integration in $(t, t+\Delta t)$, we have $i(t+\Delta t) = \frac{2C}{\Delta t}(v(t+\Delta t) - v(t)) - i(t)$, which allows us to transform a capacitor at $t+\Delta t$ into a resistor and a current source in parallel, as shown in Figure 3 (a). After transforming capacitors, the circuit contains only resistors and current sources. An example of such a circuit is shown in Figure 3 (b), where the currents i_0 and i_1 flowing through R_0 and R_1 can be represented by Ohm's law ($V = IR$). We perform nodal analysis to calculate the voltage at each node of the circuit. Specifically, at nodes v_1 and v_2 we apply Kirchhoff's current law, which states that the algebraic sum of the currents entering and exiting a node is zero. These equations can be organized as a system of linear equations and represented in the matrix/vector form $GV = I_{src}$. The voltage vector V can be calculated as $G^{-1}I_{src}$.

2.3 Related Work

The 2020 and 2021 TAU contests [1, 2] introduce two simplified versions of CCS stage delay calculation. The 2020 contest [1] uses a reduced model of RC networks, π model, consisting of a resistor and two capacitors. For this problem, Garyfallou et al. [5] develop a closed-form formula for the effective capacitance of a π-model load, which is used to compute the effective capacitance values for different waveform regions. It is efficient because of the closed-form formula but is only applicable to the simplified π-model load. On the other hand, the 2021 contest [2] provides complete RC networks instead of π models for delay calculation. Garyfallou et al. [6] propose a machine learning approach to learn the correlation between the initial computed results and the golden SPICE results for accurate delay prediction.

Existing CCS algorithms in the literature are rarely based on simulation, primarily due to the slow runtime. Also, previous research mostly focuses on CCS driver or stage delay calculation instead of graph-level analysis, because it is straightforward to obtain the graph-level results by propagating the stage delay results. By contrast, GCS-Timer uses simulation-based analysis for high accuracy and overcomes its long runtime by GPU acceleration. GCS-Timer also performs graph-based analysis, which opens up more opportunities for parallelization.

3 GCS-TIMER

In this section, we discuss the details of GCS-Timer. Section 3.1 starts with the overview of GCS-Timer. Next, we introduce our driver output current computation engine in Section 3.2, followed by our GPU-accelerated simulation kernel in Section 3.3. Lastly, in Section 3.4, we propose a precomputation technique to accelerate matrix inverse computation.

3.1 Overview

Algorithm 1 shows the overall flow of GCS-Timer, which has four phases. The first phase (line 1) precomputes the partial inverse results of the simulation matrices, which can speed up the inverse computation in phases 2-3 and will be discussed in Section 3.4. The second and third phases are both simulations (pass 1 and pass 2), and have the following two differences.

Algorithm 1 Flow of GCS-Timer

```
1:  Partial matrix inverse precomputation                ▷ End of precomputation
2:  Compute complete matrix inverses (NLDM receiver model)
3:  for i ← 0 to max_level do
4:      Simulate all stages of level i in parallel
5:  end for                                              ▷ End of pass 1
6:  Compute complete matrix inverses (CCS receiver model)
7:  Simulate all stages in parallel                      ▷ End of pass 2
8:  for i ← 0 to max_level do
9:      Propagate delay of all cell pins of level i
10: end for                                              ▷ End of delay propagation
```

- The receiver models used for pass 1 and pass 2 are the NLDM and CCS receiver models, respectively. We do not use the CCS receiver model in pass 1 because the receiver input pin slew is not available before pass 1, but is necessary to compute the CCS receiver model. To solve this problem, we use the NLDM receiver model for pass 1, which models the receiver as a constant-capacitance capacitor. With the input slew calculated in pass 1, we use the CCS receiver model in pass 2 for accurate analysis.
- Pass 1 and pass 2 have different parallelisms in stage delay calculation, as indicated in Algorithm 1. Pass 1 simulates all stages in the same level in parallel, while pass 2 simulates all stages in parallel. This is due to the availability of the stage driver input slew, which is required for simulation. For the driver of a stage, its input slew was computed when it acted as a receiver of some stages in a previous circuit level. Therefore, pass 1 needs to finish simulating one level of stages before moving on to the next level to obtain all the driver input slews for the next level. For pass 2, it uses the input slew results calculated in pass 1 and can therefore simulate all stages in parallel.

The last phase of the GCS-Timer flow is parallel delay propagation (lines 8-10 of Algorithm 1), where we propagate the delay level by level. We compute the cumulative delay of all pins in the same level simultaneously. We will not go into the details of delay propagation as it is a standard process (see [13] for details).

Algorithm 2 Driver Output Current Computation

```
1:  CCS Driver Model:
2:  a list of input transition time values, input_slews
3:  a list of output load capacitance values, output_caps
4:  a piecewise linear function i(t) for every (slew, cap) pair,
    where slew ∈ input_slews, cap ∈ output_caps
5:  function f_{i_out}(v_out, t_0)
6:      Interpolate with the driver input slew
7:      Compute v(t) from i(t) for each cap ∈ output_caps
    // The driver model at this point can be represented by g(c, v, t) = 0.
    // Given the values of any two variables, g can compute the third one.
8:      Compute c_0 from g(c_0, v_out, t_0) = 0
9:      Compute t^- from g(c_0, v_out − Δv, t^-) = 0
10:     Compute t^+ from g(c_0, v_out + Δv, t^+) = 0
11:     return i_out = c_0 × 2Δv/(t^+ − t^-)
12: end function
```

3.2 Driver Output Current Computation

Driver output current (i_{out}) computation is important for ensuring accurate driver response during simulation. Algorithm 2 shows our algorithm $f_{i_{out}}$, which computes i_{out} using the driver output voltage v_{out}, time t_0, and the CCS driver model. The driver model provides a list of input slew and output capacitance values, and an $i(t)$ function for each input slew/output capacitance pair.

$f_{i_{out}}$ starts with interpolating using the input slew of the driver input pin. Next, we compute a voltage function, $v(t)$, from $i(t)$ for each output capacitance value. After these two steps, the driver model can be represented by an implicit function $g(c, v, t) = 0$, which, given the values of any two variables (out of capacitance c, voltage v and time t), returns the value of the third variable. With this model, we compute i_{out} in lines 8-11 of Algorithm 2, according $i = C(v^+ - v^-)/(t^+ - t^-)$, where $v^\pm = v_{out} \pm \Delta v$ with their corresponding times $t^\pm$. The formula is an approximation of $i = C\frac{dv}{dt}$. In our implementation, $\Delta v = 0.01V_{dd}$.

Algorithm 3 GPU-Accelerated Simulation Kernel

Input: a stage with input slew; **Output**: slews (pass 1) or delays (pass 2)
//n: number of circuit nodes; $V^{(i)}$: voltage vector at iteration i
1: **while** not all receiver input pin voltages reach V_{dd} **do**
2: $t^{(i)} \leftarrow t^{(i-1)} + \Delta t$ ▷ Iteration i
3: **for** $j \leftarrow 1$ to n **in parallel do**
4: Compute I_{src_j}
5: **end for**
6: $i_{out} \leftarrow f_{i_{out}}(v_{out}, t^{(i)})$
7: $I_{src_1} \leftarrow I_{src_1} + i_{out}$
} Compute I_{src} in parallel
8: **for** $j \leftarrow 1$ to n **in parallel do**
9: **for** $k \leftarrow 1$ to n **do**
10: $V_j^{(i)} += G^{-1}_{j,k} I_{src_k}$
11: **end for**
12: **end for**
} Compute $V^{(i)}$ in parallel
13: **end while**
14: Compute receiver input pin slews (pass 1) or cell and net delays (pass 2)

3.3 GPU-Accelerated Simulation

Simulation is the core of GCS-Timer, which computes the voltage at each node for a sequence of timesteps $t^{(0)}, t^{(1)}, ..., t^{(end)}$. Details of our GPU-accelerated simulation are given in Algorithm 3. The simulation loop starts with an initial state of $V^{(0)} = \mathbf{0}$ at $t^{(0)}$, and iteratively computes the new $V^{(i)}$ at time $t^{(i)}$ until all voltages of the receiver input pins reach V_{dd}. Each simulation iteration computes I_{src} followed by $V^{(i)} = G^{-1} I_{src}$, as shown in Algorithm 3.

I_{src} is the current source vector and I_{src_j} is the total current entering node j from current sources, including both capacitor-transformed and driver-model current sources. In each simulation iteration, we compute the driver-model current using our i_{out}-computation engine $f_{i_{out}}(v_{out}, t)$ in Algorithm 2 (the computation of v_{out} will be discussed later). For the capacitor-transformed current sources, we use n threads to compute their current values in parallel because the computations are independent.

Next, after obtaining I_{src}, we compute $V^{(i)}$ by $G^{-1} I_{src}$, which is a matrix-vector multiplication. Again, we use n threads to compute $V^{(i)}$ in parallel with each thread adding n multiplication results. It is possible to further parallelize the loop in lines 9-11 that adds n numbers, for example, by parallel reduction. However, we do not do this for two reasons. First, we think that n^2 threads for a single simulation is not resource efficient, considering that many simulations run in parallel. It might be better to use those n^2 threads to run n simulations in parallel. Second, parallel reduction can only reduce the runtime from $O(n)$ to $O(\log n)$ with some overhead, which is not that significant in our case where n is small on average.

Lastly, we go back to discuss how to compute v_{out}, which is needed to compute i_{out}. Assuming that the driver output point is node 1, v_{out} is $V_1^{(i)}$, which is to be computed after computing I_{src}. In other words, the computations of I_{src} and $V^{(i)}$ are inter-dependent. To break the dependency, we choose $V_1^{(i-1)}$ as an initial approximation for v_{out}, and use an iterative approach in Algorithm 4 to ensure accuracy.

Algorithm 4 An Iterative Approach to Compute Accurate v_{out}

1: **for** $iter \leftarrow 0$ to max_iter **do**
2: $v_{out} \leftarrow iter > 0 \ ? \ V_1^{(i)} \ : \ V_1^{(i-1)}$
3: Compute I_{src}, Compute $V^{(i)}$ ▷ Algorithm 3
4: **end for**

We first set $v_{out} \leftarrow V_1^{(i-1)}$ and finish the simulation iteration to get $V^{(i)}$, which corresponds to $iter = 0$ in Algorithm 4. Next, we repeat the ith simulation iteration for max_iter times with $v_{out} \leftarrow V_1^{(i)}$. Approximating v_{out} by $V_1^{(i-1)}$ for $iter = 0$ is simple and reasonable because the voltage change in Δt is usually small, and the iterative approach also helps v_{out} to converge to the true v_{out}, which improves the simulation accuracy.

We can further improve the efficiency of Algorithm 4 by computing only $V_1^{(i)}$ instead of $V^{(i)}$ for $iter < max\_iter$, because all the iterations with $iter < max\_iter$ are used only to obtain a more accurate v_{out}, and only the $V^{(i)}$ computed in the last iteration with $iter = max\_iter$ is useful. In our implementation, $max\_iter = 1$.

3.4 Partial Matrix Inverse Precomputation

A key step in the simulation is to compute the voltage vector V using G^{-1}. We observe that the matrix inverses of interest are an order of magnitude more than the number of nets, resulting in extremely long runtime even with GPU acceleration. To improve the efficiency of the inverse computation, we propose a novel precomputation technique based on the observation that many matrices differ only in a small square region.

For example, in Figure 4, we have an RC network with n nodes, and m of them are connected to the receivers. This network appears in 4 stages, each of which has three different receiver capacitance computing scheme in the 2-pass simulation process. In total, there are 12 circuits containing this RC network, differing only in the receiver capacitance values.

To see this difference in the conductance matrix format, we need to first number the nodes of the RC network. If we number them starting from the receivers (i.e., the receiver input points are numbered from 1 to m), the conductance matrices differ only in the upper left $m \times m$ region. This property allows us to precompute

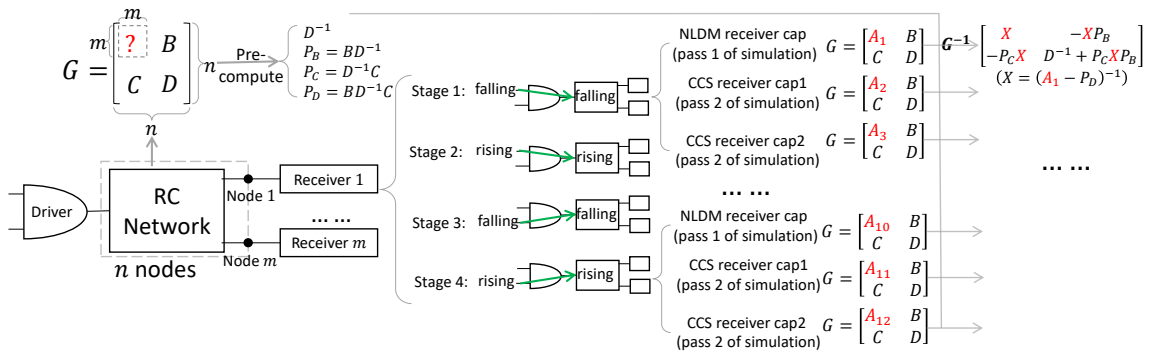

Figure 4: A partial precomputation scheme for faster matrix inverse computation. Each RC network is associated with several conductance matrices G, which are identical except for the upper-left $m \times m$ square A. We use the identical parts B, C and D to precompute D^{-1}, P_B, P_C and P_D, enabling faster inverse computation when presented with different A, according to the block matrix inversion formula.

some results of the unchanged part according to the block matrix inversion rules [3], which supports faster matrix inverse computation every time when the unknown upper-left part is given.

Specifically, for an RC network with n nodes and m receivers, if we decompose its conductance matrix G into $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$, where A is an $m \times m$ sub-matrix, its inverse can be computed by

$$\begin{bmatrix} (A - BD^{-1}C)^{-1} & -(A - BD^{-1}C)^{-1}BD^{-1} \\ -D^{-1}C(A - BD^{-1}C)^{-1} & D^{-1} + D^{-1}C(A - BD^{-1}C)^{-1}BD^{-1} \end{bmatrix}$$

according to [3]. A is initially unknown while B, C and D are available from the start. To minimize the inverse computation time, we identify the precomputable partial results of the inverse as much as possible and precompute them, which are D^{-1}, $P_B = BD^{-1}$, $P_C = D^{-1}C$ and $P_D = BD^{-1}C$. With the precomputation results, we can compute G^{-1} for a given A by computing $X = (A - P_D)^{-1}$ followed by

$$G^{-1} = \begin{bmatrix} X & -XP_B \\ -P_C X & D^{-1} + P_C X P_B \end{bmatrix}$$

, which reduces the runtime of inverse computation from $O(n^3)$ to $O(n^2 m)$. The precomputation time is also small, because the number of precomputation is much smaller than the number of inverse computation. For example, in Figure 4, we precompute only once for an RC network that has 12 matrix inverse computations during CCS delay calculation. In our experiments, the number of inverse computation is 20× more than that of the precomputation.

4 EXPERIMENTAL RESULTS

We compare GCS-Timer with an industrial standard timer ("Baseline") that has been successful in sign-off analysis. To the best of our knowledge, there are no existing open-source STA projects in support of CCS. Our experiments run on a machine with 1.90GHz CPUs and a GPU (10496 cores, 556.0 GFLOPS for FP64). GCS-Timer runs on the Ubuntu machine using only one CPU thread. Baseline runs on the CentOS machine due to commercial installation requirement, and is configured with 16 threads at which its performance saturates. Although GCS-Timer and Baseline run on different GPUs, the impact on performance is minimal because GCS-Timer mostly runs on the GPU where only one CPU thread is used for handling lightweight control-flow code.

In terms of circuit benchmarks, we synthesize the four largest arithmetic circuits from the EPFL benchmark suite under the 7nm open-source technology library ASAP7. Table 1 shows the statistics of the four circuits. The largest circuit, `Hypotenuse`, has 200K gates and 200K nets. The RC network size, in terms of the number of capacitors, ranges from 3 to 468.

4.1 Accuracy Comparison

We first evaluate the accuracy of stage delay calculation. We ask Baseline to write a SPICE deck for each stage, and use a commercial SPICE tool to generate the golden results of all stages. The accuracy results are shown in the "Stage Delay Error" column of Table 1, which is the average relative error of all cell/net arc delays compared to the golden results. Note that if the absolute error of an arc is less than $0.5ps$, we regard it as 0. This is because the delay of a small net can be very small (e.g., as small as $0.006ps$ in the circuit `Multiplier`), and therefore even an insignificant difference of $0.01ps$ would contribute a huge relative error (e.g., $0.01/0.006 = 167\%$). The 2020 and 2021 TAU contests [1, 2] have a similar way of evaluation, but they consider an absolute error of up to $2ps$ as 0. In fact, without this $0.5ps$ cutoff, the relative errors of GCS-Timer and Baseline would be 5.94% and 19.14% for circuit `Hypotenuse`, respectively. This result shows that even when the computed delay is extremely small, GCS-Timer can still achieve a superior accuracy performance over the Baseline. Additionally, we can see from Table 1 that GCS-Timer has an average stage delay error of 1.8%, which is only 2/3 the error of Baseline.

Next, we evaluate the overall accuracy of the delay propagation results of GCS-Timer. Unlike the SPICE-based evaluation method of stage delay calculation accuracy, we cannot use SPICE for evaluating the delay propagation accuracy because SPICE is not capable of performing graph-based analysis. We resort to evaluating the delay propagation accuracy by comparing the difference in arrival time values at output ports between GCS-Timer and Baseline, and show the results in the column "AT Diff" of Table 1. The average difference of the four benchmarks is 1.3%. Since GCS-Timer and

Table 1: Speed and Accuracy Comparisons of GCS-Timer and Baseline

Design	#PIs	#POs	#Gates	#Nets	RC Network Size			Runtime (s)			Speedup*	Stage Delay Error†		AT Diff‡
					Min	Avg	Max	Baseline 1 Thread	Baseline 16 Threads	GCS-Timer		Baseline	GCS-Timer	
Multiplier	128	128	26654	26782	3	12	327	20.1	4.0	1.5	13.4×/2.7×	2.1%	2.0%	2.3%
Log2	32	32	31361	31393	3	13	433	22.1	4.8	1.5	14.7×/3.2×	3.0%	1.7%	1.7%
Divisor	128	128	100955	101083	3	11	426	55.7	10.3	3.0	18.6×/3.4×	3.5%	1.9%	1.0%
Hypotenuse	256	256	205295	205551	3	11	468	131.1	32.2	9.7	13.5×/3.3×	2.3%	1.7%	0.3%
Average											15.1×/3.2×	2.7%	1.8%	1.3%

†Average cell/net delay error compared to a commercial SPICE tool
*Speedup of GCS-Timer over 1/16-threaded Baseline
‡Average difference of arrival times at output ports between GCS-Timer and Baseline

Baseline already have slightly different stage delay calculation results, it is reasonable to see a small difference in the arrival time values at the output ports between Baseline and GCS-Timer.

4.2 Speed Comparison

The "Runtime" in Table 1 compares the STA runtime between Baseline and GCS-Timer. We run Baseline twice with 1 and 16 threads (at which its performance saturates), and we run GCS-Timer with 1 GPU and 1 CPU thread. We show the speedups of GCS-Timer over the single-threaded and 16-threaded Baseline in the "Speedup" column of Table 1. On average, GCS-Timer is 15.1× and 3.2× faster than single-threaded and 16-threaded Baseline respectively, which is a significant improvement considering that GCS-Timer also achieves better accuracy. The largest speedup happens at the circuit `Divisor`, where GCS-Timer is 18.6× and 3.4× faster than the single-threaded and 16-threaded Baseline respectively.

4.3 Tradeoff Between Accuracy and Speed

The better accuracy and faster runtime results achieved by GCS-Timer give STA applications more opportunities to explore different tradeoffs between accuracy and speed. For example, we can reduce the RC network size by performing model order reduction of RC networks [18]. This decreases the complexity of RC networks, leading to faster speed yet at the cost of lower accuracy. On the other hand, we can increase simulation iterations to achieve higher accuracy yet at the cost of longer runtime. By using these two tradeoff techniques, GCS-Timer can offer various accuracy-speed tradeoffs. Figure 5 shows the tradeoffs on circuit `Hypotenuse`. As we can observe, regardless of the tradeoffs we have applied, the runtime and accuracy results are still better than the 16-threaded Baseline.

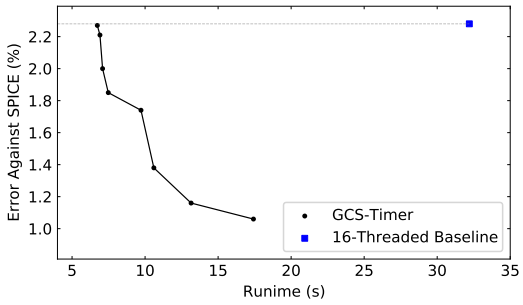

Figure 5: Tradeoffs of GCS-Timer achieved by reducing RC networks or increasing simulation iterations (`Hypotenuse`).

5 CONCLUSIONS

In this paper, we have introduced GCS-Timer, a GPU-accelerated STA algorithm with a specific focus on the CCS model. To achieve efficient and accurate CCS-based timing analysis, we design a fast and accurate GPU acceleration framework. Under this framework, we further develop an efficient CCS delay calculation engine on GPU. We also propose a precomputation technique for fast matrix inverse calculation. Compared with a 16-threaded industrial standard timer, GCS-Timer achieves an average of 3.2× faster analysis runtime with better accuracy results on a set of real circuit designs.

REFERENCES

[1] 2020. Tau 2020 contest. https://www.tauworkshop.com/2020/contest.shtml.
[2] 2021. Tau 2021 contest. https://www.tauworkshop.com/2021/contest.shtml.
[3] 2023. Block matrix — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Block_matrix [Online; accessed 16-November-2023].
[4] OpenSTA. https://github.com/The-OpenROAD-Project/OpenSTA
[5] Dimitrios Garyfallou, Stavros Simoglou, Nikolaos Sketopoulos, Charalampos Antoniadis, Christos P. Sotiriou, Nestor Evmorfopoulos, and George Stamoulis. 2021. Gate Delay Estimation With Library Compatible Current Source Models and Effective Capacitance. *IEEE TVLSI* 29, 5 (2021), 962–972.
[6] Dimitrios Garyfallou, Anastasis Vagenas, Charalampos Antoniadis, Yehia Massoud, and George Stamoulis. 2022. Leveraging Machine Learning for Gate-Level Timing Estimation Using Current Source Models and Effective Capacitance. In *Proc. ACM GLSVLSI*.
[7] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *Proc. IEEE/ACM ICCAD*.
[8] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *Proc. ACM/IEEE DAC*.
[9] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis. In *Proc. IEEE/ACM ICCAD*.
[10] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* 40, 4 (2021), 776–789. https://doi.org/10.1109/TCAD.2020.3007319
[11] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++. In *IEEE TPDS*. 974–983.
[12] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TCAD* 33, 6 (2022), 1303–1320.
[13] Tsung-Wei Huang and Martin D. F. Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *Proc. IEEE/ACM ICCAD*.
[14] Shiju Lin, Jinwei Liu, Tianji Liu, Martin D. F. Wong, and Evangeline F. Y. Young. 2022. NovelRewrite: node-level parallel AIG rewriting. In *Proc. ACM/IEEE DAC*.
[15] Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. 2023. GAMER: GPU-Accelerated Maze Routing. *IEEE TCAD* 42, 2 (2023), 583–593.
[16] Shiju Lin and Martin D. F. Wong. 2022. Superfast Full-Scale GPU-Accelerated Global Routing. In *Proc. IEEE/ACM ICCAD*.
[17] Lawrence Pillage. 1998. *Electronic Circuit & System Simulation Methods (SRE)* (1 ed.). McGraw-Hill, Inc., USA.
[18] Yangfeng Su, Fan Yang, and Xuan Zeng. 2012. AMOR: An Efficient Aggregating Based Model Order Reduction Method for Many-Terminal Interconnect Circuits. In *Proc. ACM/IEEE DAC*.