Heterogeneous Static Timing Analysis with Advanced Delay Calculator

Zizheng Guo[1,2], Tsung-Wei Huang[4], Zhou Jin[5], Cheng Zhuo[6], 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
[4]The University of Wisconsin at Madison [5]China University of Petroleum-Beijing [6]Zhejiang University

***Abstract*—Static timing analysis (STA) in advanced technology nodes encounter many new challenges in analysis accuracy and speed efficiency. To accurately model complex interconnect networks, existing timers have leveraged reduced-order models with effective capacitance to design advanced delay calculation algorithms. However, the iterative nature of these algorithms makes them extremely time-consuming to use in a timer, significantly limiting their capability in many timing-driven applications. To overcome this challenge, we propose a novel GPU-accelerated delay calculator that targets Arnoldi-based model order reduction with an effective capacitance algorithm. We design efficient numerical kernels for batched nodal analysis model construction, LU decomposition, Krylov subspace calculation, eigenvalue decomposition, and Newton-Raphson iteration. Compared with two industrial standard timers, PrimeTime and OpenSTA, we achieve a strong correlation with up to 7.27× and 14.03× speed-up, respectively.**

I. Introduction

Static timing analysis (STA) is a critical step in the overall design flow because it validate the timing behaviors of a circuit designs [1]. To accurately track the timing behaviors, several delay models have been proposed over the last decades. Among various delay models, existing timers have widely used Elmore delay model and non-linear delay model (NLDM) to calculate the net delay and gate delay, respectively. While Elmore and NLDM are advantageous in their computational efficiency, they cannot accurately model delays in advanced nodes (e.g., beyond 45nm). Specifically, Elmore model applies only a first-order approximation to the interconnect delay. The approximation is limited to simple tree-based RC structures and cannot handle complex lumped RC networks. Likewise, NLDM assumes a simple sum of total capacitance in an RC network. This estimation can overestimate the delay and slew of a gate due to resistive shielding effect.

To address this problem, industrial standard timers, such as PrimeTime and OpenSTA [2], incorporate advanced delay calculators using different model-order-reduction (MOR) techniques and effective capacitance formulas. For instance, the open-source timer, OpenSTA, incorporates Arnoldi algorithm to calculate interconnect delays with a custom capacitance formula. Despite more accurate results, the iterative process of Arnoldi algorithm incurs significant runtime cost for solving many different linear systems, as shown in Figure 1. For example, switching from Elmore to Arnoldi reduces the delay prediction error from 25.5% to 10.8% on a large design under 14nm process. However, it slows down OpenSTA by 2.55×. Even with multi-threading, OpenSTA can only speed up the process up to only 2.23×, and the results largely saturated at 16 cores [3], [4], [5], [6]. This significant runtime cost has largely limited the use of Arnoldi algorithm in many timing-driven optimization flows.

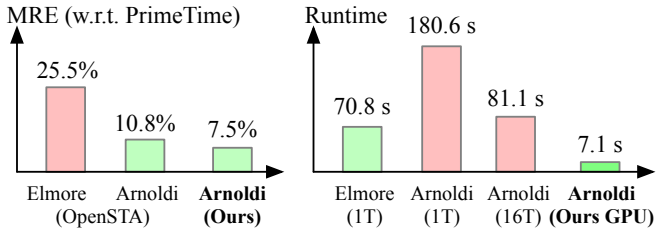


Fig. 1: Arnoldi delay models have significantly less mean relative error (MRE) in delay calculation in advanced nodes compared to Elmore models. However, it introduces significant runtime overhead and challenges in parallelization. Data collected by analyzing a million-gate industrial design `netcard` [10].

To achieve better runtime scalability, CPU-GPU heterogeneous computing techniques have been utilized to solve timing analysis tasks [7], [8], [9]. While these works are limited to Elmore model, their methods have inspired us to accelerate Arnoldi algorithm using CPU-GPU heterogeneous parallelism. Specifically, Arnoldi algorithm involves many linear algebra operations, such as dense/sparse matrix multiplication and decomposition, that can significantly benefit from the massive parallelism of GPU. However, designing a GPU-accelerated Arnoldi algorithm is very **challenging** for three reasons: **(1)** The parallelism inside timing analysis is constrained by the topology of the circuit graph, making it challenging to design efficient parallel decomposition strategies. **(2)** The amount of matrix algebras vary by the degrees of different nets, which introduces severe workload imbalance between different GPU threads. **(3)** Moreover, the iterative procedures for effective capacitance calculation lead to great divergence in computation patterns which are hard to predict and optimize.

To overcome the above challenges, we propose a novel GPU-accelerated delay calculator targeting Arnoldi model order reduction and effective capacitance algorithm. We summarize our technical contributions as follows:

*Corresponding author: Yibo Lin (yibolin@pku.edu.cn). This work is supported in part by the Natural Science Foundation of Beijing, China (Grant No. Z230002), the 111 project (B18001), National Key R&D Program of China (Grant No. 2022YFB4400400), NSFC (Grant No. 62204265), and National Science Foundation in the US under grants CCF-2349141, CCF-2349582, OAC-2349143, and TI-2349144.

1) We build a new GPU-accelerated delay calculator for advanced interconnect modeling. We combine Arnoldi model order reduction and effective capacitance computation to an accurate, efficient delay calculator.
2) We design powerful GPU kernels to accelerate all numerical tasks in advanced interconnect modeling. The tasks include batched nodal analysis construction, LU factorization, Krylov subspace calculation, eigenvalue decomposition, and Newton-Raphson iterations.
3) We integrate our delay calculator into a fully GPU-accelerated STA engine and achieve up to 7.27× speed-up over PrimeTime and 14.03× speed-up over OpenSTA on large circuit designs. The accuracy results of our timer are strongly correlated to that of PrimeTime.

The rest of this paper is organized as follows. Section II introduces the problem formulation of STA and delay calculation. Section III presents details of our GPU-accelerated delay calculator. Section IV demonstrated the experimental results. Finally, Section V concludes the paper.

II. Preliminaries

STA engines regard a circuit as a directed acyclic graph (DAG). As shown in Figure 2, this DAG represents the directions of signal transmission between pins and arcs. There are two kinds of timing arcs, net arcs and cell arcs representing interconnect and logic cells (i.e., gates), respectively. Both kinds of arcs introduce delays during the trip of a signal. An STA engine first computes the delays of all net arcs and cell arcs according to physical information (delay calculation), and then extracts the worst negative slack (WNS), total negative slack (TNS), and a list of timing-critical paths according to timing constraints (path searching).

The accuracy and speed of an STA engine are directly determined by the models used in delay calculation. The basic problem instance of delay calculation involves a logic cell driving a series of downstream cells through a metal interconnect between them. Both the distribution of lumped resistors and capacitors (RC) in the interconnect and the behavior of the driving cell affect the delays of net arcs and cell arcs as a whole. Moreover, the delays are also affected by the upstream voltage transition time (i.e., slew). The central task of delay calculation in an STA engine is to propagate the slew throughout the DAG and compute the arc delays under best-case and worst-case scenarios.

To characterize the behavior of driving cells, the non-linear delay model (NLDM) is used to generate a set of look-up tables (LUTs). Each LUT item gives the cell delay with a lumped capacitive load attached to the cell output pin. In real scenarios, the capacitor is replaced with a complex RC interconnect. The capacitances far from the driving cell will have a weak effect on cell delay due to the presence of resistances along the way. Modern signoff-level STA engines like [2] incorporate complex and iterative processes to model the interaction between cells and RC interconnects. These processes are difficult to implement due to their black-box nature, and even more difficult to accelerate due to their inherent complexity in numerical algorithms compared to simple models like Elmore delay and total capacitances [10], [11].

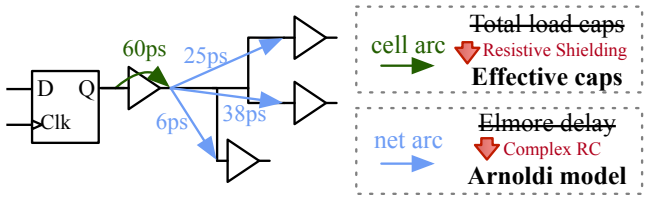


Fig. 2: A circuit graph and two types of timing arcs.

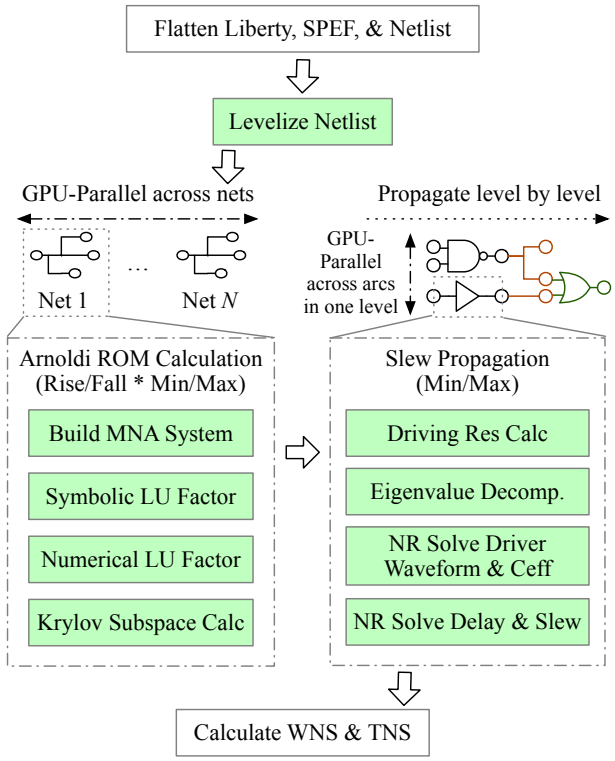


Fig. 3: Overview of computation tasks in our heterogeneous STA engine. The boxes in green indicate GPU-accelerated tasks. The flattening and levelization algorithms follow [7].

III. Algorithms

Figure 3 shows a complete task flow of our proposed heterogeneous STA engine with Arnoldi delay calculator. To maximize the parallelism and solve **Challenge (1)** in Section I, we divide the delay calculation task into two main steps: Arnoldi reduced-order model (ROM) calculation and slew propagation, with different parallelization strategies. The Arnoldi ROM calculation step consists of 4 substeps: batched modified nodal analysis (MNA) system construction, symbolic and numerical LU factorization, and Krylov subspace calculation. We parallelize their computation across different nets. The slew propagation step is also divided into 4 substeps: driving resistance calculation, eigenvalue decomposition, Newton-Raphson (NR) iterations for driver waveform and effective capacitance, and finally the delay and slew calculation. They are parallelized within every logic level of the STA graph. Following subsections describe the details of each step, our engineering effort orchestrating such a system to improve delay calculation accuracy, and our different acceleration strategies for the various numerical algorithms and computation patterns involved.

A. Batched Arnoldi ROM Calculation

Model order reduction is at the core of advanced delay calculation. It serves to accurately analyze the behavior of large RC interconnects with small approximate linear systems. In this work, we use the coordinate-transformed Arnoldi algorithm [12] as our model order reduction engine because of its guaranteed stability in the resulting systems. For simplicity, we use examples of a single net to explain the algorithms, but it is batched when considering all nets.

1) Building a MNA System: Our first task is to represent an RC interconnect with a linear system using modified nodal analysis (MNA). Specifically, the resulting system has the form

$$Cv' + Gv = e_1 \cdot u, \tag{1}$$

where we assume the interconnect has n nodes, the first pin is the driver, $v = v(t) \in \mathbb{R}^n$ is the voltage waveform of all pins, and $u = u(t)$ is the external input. C and G are sparse $n \times n$ matrices representing nodal capacitances and conductances. For our RC interconnect case, C and G are both real symmetric.

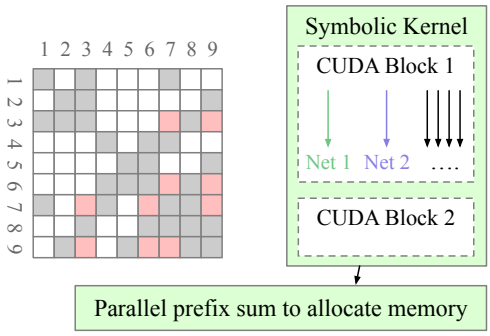
Our first task, as presented in Algorithm 1, is to construct C and G matrices according to SPICE stamps (lines 7–9) using the parasitic annotations. We store the resulting C and G in compressed sparse column (CSC) format for later steps. The careful use of radix sort and in-place CSC nonzeros construction makes it easy to port to GPUs.

Algorithm 1: Batched MNA construction.

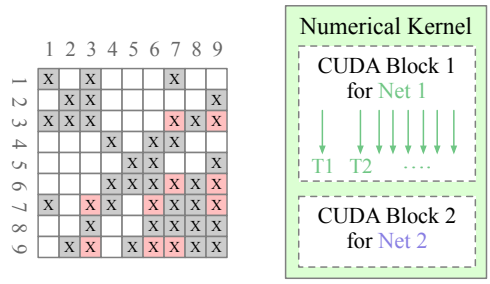
1 **GPU Parallel for** *all nets* **do**
  Input: Parasitic elements array $elem$.
2 Initialize CSC arrays Gnnz, Gp, Gv, Cnnz, Cp, Cv;
3 Accumulate number of nonzeros on each row;
4 Do in-place prefix sum on Gnnz and Cnnz;
5 Radix sort $elem$ by (a, b) for R or C interconnect $a \leftrightarrow b$ and $a < b$ (first by b, then stable sort by a);
6 Append nonzeros for all elements in order:
7 For resistance R between $a \leftrightarrow b$, insert element $G_{a,b}$-=$1/R$, $G_{a,a}, G_{b,b}$+=$1/R$;
8 For coupling capacitance C between $a \leftrightarrow b$, insert $C_{a,b}$,-=C, $C_{a,a}, C_{b,b}$+=C;
9 For grounded capacitance C on a, insert $C_{a,a}$+=C;
10 Append transpose to the CSC arrays;

2) Sparse LU (LDL) Factorization: The reduced-order system computation involves solving the equation $Gx = y$ many times. Calculating the inverse of G is discouraged in practice because it destroys its sparsity. Instead, we perform a sparse LU factorization on G similar to a SPICE simulator. Thanks to the symmetric property of G, we can write it as: $G = LDL^{\mathrm{T}}$. A sparse LU factorization involves two stages: symbolic and numerical factorization. Symbolic factorization computes the locations of nonzero elements (i.e., fill-ins) in L and numerical factorization determines the concrete values of them.

Sparse LU factorization of very large ($n \geq 10^5$) matrix systems has been accelerated using GPUs to design heterogeneous SPICE simulators [13], [14], [15]. On the contrary, we deal with millions of matrices with diverse sizes ($10 \leq n \leq 4,000$)



(a) Symbolic factorization



(b) Numerical factorization

Fig. 4: Illustration of different parallelization strategies for batched symbolic and numerical LU decomposition.

which leads to imbalanced workload and insufficient parallelism (**Challenge (2)** as stated in Section I).

As a result, we incorporate different parallelization strategies for the two stages in our STA engine, as presented in Figure 4. Symbolic factorization is mostly sequential, so we assign one GPU thread for every net (i.e., every matrix G). After symbolic factorization, we collect the number of nonzeros through a parallel prefix sum for numerical memory allocation. Numerical factorization is computation intensive and involves vector operations, so we assign one GPU block (256 threads) for every net. Our strategy turns out to be a good combination of both intra-task and inter-task parallelism.

3) Krylov Subspace through Arnoldi Process: We compute the Krylov subspace of given interconnect through Algorithm 1 in [12]. We fix the order to be 4 observing this is a good trade-off between speed and accuracy. The results are two matrices $H_q \in \mathbb{R}^{4\times4}$ and $U_q \in \mathbb{R}^{4\times n}$. During the calculation on GPU, we assign a block of threads to compute the algebraic operations involved in [12], including:

1) Solve $Gx = y$ with the sparse decomposition $G = LDL^{\mathrm{T}}$, using parallel forward substitution and backward substitution.
2) Calculate the sparse matrix-vector product $y = Cx$.
3) Calculate the dot product of two vectors in $\mathbb{R}^n$.

During the calculation, we need block-wise reduction operations. In practice, we implement the operations using CUDA shared memory

B. Batched Slew Propagation

To model the interaction between nets and driving cells, and propagate the signal transition time, we follow the method proposed by Dartu, Menezes, and Pileggi [16] (DMP method)

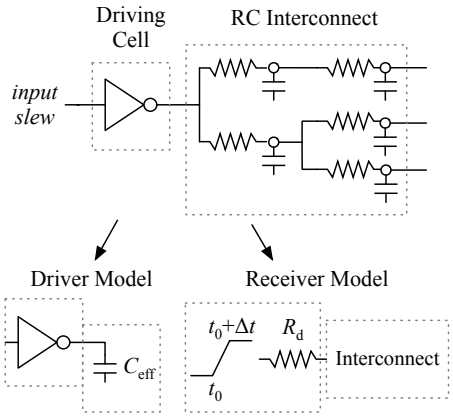


Fig. 5: Modeling the interaction between a driver cell and a receiver RC interconnect. A total of 4 parameters are included: $R_d, t_0, \Delta t$, and $C_{\texttt{eff}}$.

and re-derived their method to our reduced-order model and heterogeneous framework. Figure 5 shows the basic idea of the DMP method. On the cell side, the interconnect is regarded as an effective capacitance $C_{\texttt{eff}}$. On the net side, the cell is modeled as a ramp shape driver voltage waveform $(t_0, t_0 + \Delta t)$ and a driving resistance R_d. Our goal is to solve the parameters $R_d, t_0, \Delta t$, and $C_{\texttt{eff}}$ involved between cell drivers and interconnects. For simplicity, we use examples of a single cell arc to explain the algorithms, but it is batched when considering all cell arcs in every logic level.

1) Driving Resistance: We use the following Equation (2) to approximate the driving resistance.

$$R_d = \frac{slew}{totalcap \cdot \log \frac{thres_{slew2}}{thres_{slew1}}} \tag{2}$$

It is an adaptation of the Equation (22) in [16] and is derived from an exponential fit of the output waveform between two slew threshold points. The *slew* in Equation (2) comes from an initial NLDM table lookup using input slew and total capacitance.

2) Eigenvalue Decomposition for Poles and Residues: By obtaining the driving resistance, we update the interconnect system and obtain a new $H_q \in \mathbb{R}^{4\times4}$. The eigenvalue decomposition of H_q will then give the exact solution of this reduced-order transient system. Specifically, the transfer function between driver and all nodes is written as

$$\frac{X(s)}{u(s)} \approx U_q(I - H_q s)^{-1} e_1 \sqrt{\|G^{-1}e_1/R_d\|_C^2}, \tag{3}$$

which corresponds to Equation (14) in [12]. Substituting an eigenvalue decomposition $H_q = M\Lambda M^{\mathrm{T}}$, we get

$$\frac{X(s)}{u(s)} \approx U_q M(I - \Lambda s)^{-1} M^{\mathrm{T}} e_1 \sqrt{\|G^{-1}e_1/R_d\|_C^2}, \tag{4}$$

where $(I - \Lambda s)^{-1}$ is a trivial inverse of a diagonal matrix.

Because we have a pure RC interconnect, H_q is guaranteed to be tridiagonal symmetric. As a result, we use the classic Jacobian iteration method to find the eigenvalues and eigenvectors. We represent the step-response voltage waveform solution at one node as a set of poles p_i and residues r_i $(i = 1, 2, 3, 4)$. The waveform in Laplace domain and time domain are respectively,

$$\mathcal{L}(v(t)) = \sum_{i=1}^{4} \frac{r_i p_i}{s(p_i - s)}, \tag{5}$$

$$v(t) = 1 - \sum_{i=1}^{4} r_i e^{p_i t}. \tag{6}$$

We use Algorithm 2 to calculate the poles and residues, which implements Equation (4). On GPU, we assign 1 thread to calculate the poles (line 1) of every net, and a block of threads to calculate the residues (line 3) of all pins in the net.

Algorithm 2: Compute poles and residues of one net.

Input: $H_q \in \mathbb{R}^{4\times4}$ and $U_q \in \mathbb{R}^{4\times n}$
Input: $H_q = M\Lambda M^{\mathrm{T}}$ where $\Lambda = \texttt{diag}\{\lambda_1, ..., \lambda_4\}$
1 **for** $i = 1, 2, 3, 4$ **do**
2 $p_i \leftarrow -1/\lambda_i$;
3 **GPU Parallel for** $j = 1, 2, ..., n$ **do**
4 **for** $i = 1, 2, 3, 4$ **do**
5 $r_i^{(j)} \leftarrow 0$;
6 **for** $k = 1, 2, 3, 4$ **do**
7 $r_i^{(j)} \leftarrow r_i^{(j)} + U_q[k, j]M[k, i]$;
8 $r_i^{(j)} \leftarrow r_i^{(j)} \cdot M[1, i]$;
9 Normalize $r_{1,2,3,4}^{(j)}$ so that they sum to 1;

3) NR Iteration for Driver Parameters: Computing the driver parameters $C_{\texttt{eff}}, \Delta t, t_0$ is the central task in delay calculation. Because the delay of nets and driving cells have subtle influence on each other, driver parameters are solved iteratively. To minimize the threading divergence (**Challenge (3)** in Section I), we need an algorithm with fast convergence rate. An ideal set of parameters should satisfy the following Equation (7)[1], adapted from Equation (12) in [16].

$$\begin{cases} y(delay(C_{\texttt{eff}}), t_0, \Delta t) = 0.5, \\ y(delay(C_{\texttt{eff}}) - \frac{1}{2} \cdot slew(C_{\texttt{eff}}), t_0, \Delta t) = 0.2, \\ Q_{C_{\texttt{eff}}}(\Delta t) = Q_{\texttt{ROM}}(\Delta t). \end{cases} \tag{7}$$

Basically, Equation (7) says that the waveform at the root node under R_d should match the delay and first threshold points as specified in NLDM library. Furthermore, the current drawn by the effective capacitance $C_{\texttt{eff}}$ should be the same as the actual interconnect during the driver ramp Δt. The currents $Q_{C_{\texttt{eff}}}(\Delta t)$ and $Q_{\texttt{ROM}}(\Delta t)$ are derived respectively in Equations (9) and (11).

$$v_{C_{\texttt{eff}}}(t) = \frac{1}{\Delta t} \int_0^t 1 - e^{-\frac{t_1}{R_d C_{\texttt{eff}}}} \, \mathrm{d}t_1 \tag{8}$$

$$\begin{aligned} Q_{C_{\texttt{eff}}}(\Delta t) &= \int_0^{\Delta t} \frac{1}{R_d} \left(\frac{t}{\Delta t} - v_{C_{\texttt{eff}}}(t) \right) \mathrm{d}t \\ &= C_{\texttt{eff}} + \frac{C_{\texttt{eff}}^2}{\Delta t} \left(-1 + e^{-\frac{\Delta t}{C_{\texttt{eff}} R_d}} \right) R_d \end{aligned} \tag{9}$$

[1]In practice, 0.5 and 0.2 in Equation (7) should be replaced with $thres_{delay}$ and $thres_{slew1}$ defined in Liberty library.

$$v_{\texttt{ROM}}(t) = \frac{1}{\Delta t}\int_0^t 1 - \sum_{i=1}^{4} r_i e^{p_i t_1}\,\mathrm{d}t_1 \tag{10}$$

$$\begin{aligned} Q_{\texttt{ROM}}(\Delta t) &= \int_0^{\Delta t} \frac{1}{R_d}\left(\frac{t}{\Delta t} - v_{\texttt{ROM}}(t)\right)\mathrm{d}t \\ &= \frac{1}{R_d}\sum_{i=1}^{4} r_i \frac{-1 + e^{p_i \Delta t} - p_i \Delta t}{p_i^2 \Delta t} \end{aligned} \tag{11}$$

To model the voltage waveform under a ramp input $(t_0, t_0 + \Delta t)$ (see Figure 5) instead of a step input, we perform an integration on Equation (6). We also need the gradient of waveform points over the time point and the ramp duration. Algorithm 3 calculates a voltage at time t on the waveform given ramp input $(0, \Delta t)$, as well as the gradient over time t and ramp duration Δt.

Algorithm 3: Waveform gradient (1 GPU thread).

1 **def** *calc_waveform_grad*($t, \Delta t, p_i, r_i$)**:**
2 **if** $t < 0$ **then**
3 **return** y=0, dy/dt=0, dy/dΔt=0;
4 $t_1 \leftarrow \max(0,\ t - \Delta t)$;
5 $y \leftarrow \frac{1}{\Delta t}\left(t - t_1 - \sum_{i=1}^{4} \frac{r_i}{p_i}(e^{t p_i} - e^{t_1 p_i})\right)$;
6 dy/d$\Delta t \leftarrow -y/\Delta t$;
7 dy/d$t \leftarrow \frac{1}{\Delta t}\left(1 - \sum_{i=1}^{4} r_i e^{t p_i}\right)$;
8 **if** $t \geq \Delta t$ **then**
9 $g \leftarrow \frac{1}{\Delta t}\left(1 - \sum_{i=1}^{4} r_i e^{(t-\Delta t) p_i}\right)$;
10 dy/d$\Delta t \leftarrow$dy/d$\Delta t + g$;
11 dy/d$t \leftarrow$dy/d$t - g$;
12 **return** y, dy/dt, dy/dΔt;

Given the waveform gradient and currents, we solve Equation (7) using a two-level approach, as presented in Algorithm 4. In the inner loop (lines 7–12), we solve the first two waveform-related equalities using Newton-Raphson iteration. In the outer loop, we solve $C_{\texttt{eff}}$. By setting $Q_{C_{\texttt{eff}}}(\Delta t) = Q_{\texttt{ROM}}(\Delta t)$ (see Equations (9) and (11)) and discarding terms above third order, we solve a plain quadratic equation analytically (lines 13–14). All parameters usually converge within 10–30 inner iterations.

4) NR Iteration for Interconnect Delay and Slew: Finally after solving the driving parameters, we have determined the voltage waveform. Similar to Algorithm 4 lines 7–12, we use NR iteration to find the time points at 50%, 20%, and 80% voltage and compute the delay and slew according to their definitions. This is the last step in one level of delay calculation and slew propagation. By repeatedly propagating slew on the levelized circuit graph, we solve the delays of all timing arcs.

IV. Experimental Results

We implemented our heterogeneous STA engine using Rust, C++, and CUDA. To evaluate the accuracy and efficiency on advanced nodes, we use the TAU 2015 contest benchmarks [10] and re-synthesized all benchmark netlists under an industrial 14nm technology. The sizes of all netlists are listed in Table I. The largest designs `leon2` and `netcard` have millions of

Algorithm 4: Batched driving parameter solver.

1 **GPU Parallel for** *all cell arcs in current level* **do**
2 $C_{\texttt{eff}} \leftarrow$ total capacitance;
3 $\Delta t \leftarrow$ initial slew$/(thres_{slew2} - thres_{slew1})$;
4 $t_0 \leftarrow$ initial delay$+\log(thres_{delay}) R_d C_{\texttt{eff}} - \Delta t/2$;
5 **while** $C_{\texttt{eff}}$ *not converged* **do**
6 $delay, slew \leftarrow$ lookup NLDM using $C_{\texttt{eff}}$;
7 **while** $t_0, \Delta t$ *not converged* **do**
8 f_1, df_1/dt, df_1/d$\Delta t \leftarrow$ calc_waveform_grad($delay - t_0$, Δt) - 0.5;
9 f_2, df_2/dt, df_2/d$\Delta t \leftarrow$ calc_waveform_grad($delay - slew/2 - t_0$, Δt) - 0.2;
10 $\delta \leftarrow \begin{pmatrix} \mathrm{d}f_1/\mathrm{d}\Delta t & -\mathrm{d}f_1/\mathrm{d}t \\ \mathrm{d}f_2/\mathrm{d}\Delta t & -\mathrm{d}f_2/\mathrm{d}t \end{pmatrix}^{-1} \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}$;
11 $\Delta t \leftarrow \Delta t - \delta_0$;
12 $t_0 \leftarrow t_0 - \delta_1$;
13 $quad \leftarrow \frac{1}{\Delta t} R_d \left(-1 + e^{-\Delta t/(C_{\texttt{eff}} R_d)}\right)$;
14 $C_{\texttt{eff}} \leftarrow \frac{-1+\sqrt{1+4Q_{\texttt{ROM}}(\Delta t)\,quad}}{2\,quad}$;

TABLE I: Benchmark statistics.

Benchmark	#Gates	#Nets	#Pins	#RC Nodes
aes_core	22938	23199	66221	413058
b19_iccad	255278	255300	776320	4416480
des_perf_ispd	138878	139112	371587	2095933
edit_dist_ispd	147650	150212	416609	2555873
fft_ispd	38158	39184	116139	631491
leon2_iccad	1616369	1616984	4178874	22450936
leon3mp_iccad	1247725	1247979	3267993	17647115
matrix_mult_ispd	164040	167242	475186	2752675
mgc_edit_dist_iccad	161692	164254	444693	2431266
mgc_matrix_mult_iccad	171282	174484	489670	2710343
netcard_iccad	1496719	1498555	3901343	21023425
pci_bridge32_ispd	40790	40950	108172	577083
vga_lcd_iccad	259067	259152	662179	3539206

logic gates, and their most complex nets include thousands of RC nodes. We made a fair comparison with two commercial STA engines supporting signoff-accurate delay calculation, Synopsys PrimeTime and OpenSTA [2]. Both baseline STA engines are configured to use Arnoldi-based net modeling and effective capacitances. PrimeTime is regarded as the golden result. We conducted all experiments on an Ubuntu Linux machine with 32 CPU cores on 2.9 GHz, 64 GB RAM, and one Nvidia RTX 3090 GPU. We measured the average wall-clock runtime for the entire `report_timing` calls of all 3 STA engines. We measured the delay calculation accuracy by comparing the standard delay format (SDF) output of the STA engines, which include the delay of all net arcs and cell arcs.

Table II gives a comprehensive efficiency and accuracy comparison between our timer, PrimeTime, and OpenSTA.

1) Accuracy: Regarding both mean absolute error (MAE) and the R^2 score on delay calculation, our timer outperforms OpenSTA and shows a significantly better match to the golden PrimeTime. We have reached an average R^2 score of 0.985, better than OpenSTA's 0.973. Meanwhile, OpenSTA gives inferior R^2 matches (0.92 to 0.93) on some designs like `mgc_edit_dist` and `mgc_matrix_mult` whereas our

TABLE II: Overall efficiency and accuracy comparison between PrimeTime, OpenSTA, and our timer.

Benchmark	PrimeTime (16C)				OpenSTA (16C)				Ours (16C + GPU)			
	Runtime	RTR	MAE	R^2	Runtime	RTR	MAE	R^2	Runtime	RTR	MAE	R^2
aes_core	670.62	1.62	0.00	1.000	1330.98	3.22	0.18	0.995	413.64	1.00	0.24	0.988
b19_iccad	8596.88	2.67	0.00	1.000	16580.22	5.15	10.02	0.977	3218.93	1.00	4.84	0.985
des_perf_ispd	3061.67	2.86	0.00	1.000	7153.86	6.68	0.65	0.981	1071.39	1.00	0.54	0.993
edit_dist_ispd	4514.95	2.43	0.00	1.000	12071.41	6.50	1.05	0.978	1857.63	1.00	1.05	0.984
fft_ispd	1188.18	1.33	0.00	1.000	2858.55	3.19	0.68	0.979	895.49	1.00	0.91	0.973
leon2_iccad	52428.27	7.27	0.00	1.000	101163.31	14.03	6.01	0.963	7208.56	1.00	1.76	0.993
leon3mp_iccad	41380.85	5.87	0.00	1.000	79411.68	11.27	10.00	0.976	7043.75	1.00	4.77	0.985
matrix_mult_ispd	3889.05	1.91	0.00	1.000	10868.02	5.32	0.90	0.983	2041.16	1.00	1.04	0.976
mgc_edit_dist_iccad	6205.73	2.27	0.00	1.000	13904.51	5.09	3.58	0.921	2729.06	1.00	2.40	0.988
mgc_matrix_mult_iccad	6508.40	1.98	0.00	1.000	12526.86	3.81	2.70	0.932	3289.62	1.00	2.40	0.969
netcard_iccad	43520.28	6.14	0.00	1.000	81761.80	11.53	9.00	0.984	7093.04	1.00	5.78	0.985
pci_bridge32_ispd	1078.99	1.85	0.00	1.000	2160.30	3.70	1.34	0.992	584.43	1.00	0.66	0.998
vga_lcd_iccad	7441.11	2.99	0.00	1.000	12831.15	5.16	6.41	0.988	2485.26	1.00	4.00	0.984
Average	13883.46	3.17	0.00	1.000	27278.66	6.51	4.04	0.973	3071.69	1.00	2.34	0.985

Runtime: in ms. **RTR**: runtime ratio over ours. **MAE**: mean absolute error in ps over PrimeTime. R^2: the R^2 regression score over PrimeTime.

timer gives more stable delay calculation results. Our timer has greatly reduced the MAE of delay calculation, especially on large and complex designs. For example, OpenSTA has an average of 6.01ps error compared to PrimeTime on `leon2`, whereas our timer have reduced the error to 1.76ps. Our average MAE among all designs is 2.34ps, which is 42% smaller than OpenSTA's 4.04ps. These accuracy results prove the correctness and effectiveness of our proposed Arnoldi delay calculation and effective capacitance iteration framework, and show that the resulting accuracy of our timer is comparable to leading commercial tools.

2) Efficiency: We configured both PrimeTime and OpenSTA to use 16 CPU threads during analysis, and our timer uses 16 CPU threads and 1 GPU. Our runtime has outperformed both PrimeTime and OpenSTA among all benchmarks we have tested. Considering all large and small designs, we are 3.17× faster than PrimeTime and 6.51× faster than OpenSTA on average. We achieve larger speed-up on larger designs. For example, on the largest design `leon2`, we are 7.27× faster than PrimeTime and 14.03× faster than OpenSTA, completing the analysis using only 7.2 seconds instead of minutes. These runtime results prove the efficiency of our heterogeneous acceleration and scheduling for the numerical algorithms in model order reduction and effective capacitance calculation.

V. Conclusion

This paper proposes the first GPU-accelerated delay calculator for advanced delay modeling. We have built a comprehensive timing analysis system with comparable accuracy to commercial tools like PrimeTime and OpenSTA. We achieve up to 7.27× speed-up over PrimeTime and 14.03× over OpenSTA given the maximum CPU parallelism. Our STA engine provides even closer results to PrimeTime compared with OpenSTA.

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] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.

[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] 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.

[6] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, vol. 33, no. 6, 2022, pp. 1303–1320.

[7] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.

[8] 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.

[9] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *Proc. ICCAD*. ACM, 2021.

[10] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.

[11] 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.

[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] J. Zhao, Y. Wen, Y. Luo, Z. Jin, W. Liu, and Z. Zhou, "SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs," in *Proc. DAC*. San Francisco, CA, USA: IEEE, 2021, pp. 37–42.

[14] S. Peng and S. X.-D. Tan, "GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation," *IEEE Design & Test*, vol. 37, no. 3, pp. 78–90, 2020.

[15] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 3, pp. 786–795, 2015.

[16] F. Dartu, N. Menezes, and L. Pileggi, "Performance computation for precharacterized CMOS gates with RC loads," *IEEE TCAD*, vol. 15, no. 5, pp. 544–553, 1996.