General-Purpose Gate-Level Simulation with Partition-Agnostic Parallelism

Zizheng Guo[1], Zuodong Zhang[1], Xun Jiang[1], Wuxi Li[3], Yibo Lin[1,2*], Runsheng Wang[1,2], Ru Huang[1,2]
[1]School of Integrated Circuits, Peking University, Beijing, China
[2]Institute of Electronic Design Automation, Peking University, Wuxi, China [3]AMD Inc.
{gzz,yibolin,r.wang,ruhuang}@pku.edu.cn

***Abstract*—Gate-level simulation with delay annotation is a both critical and time-consuming task in the circuit design flow. It is highly nontrivial to parallelize a simulation process, especially on designs with arbitrary general-purpose sequential elements such as latches, gated clocks, and scan chains. Current works on parallelizing gate-level simulation are fundamentally incompatible with these design elements and are highly reliant on circuit partitioning to achieve the best performance. In this paper, we propose a general-purpose gate-level simulation engine with partition-agnostic parallelism. We propose a general sequential behavior encoding technique and a fast event scheduling algorithm for general-purpose simulation tasks. Experimental results have shown up to 30× speed-up over commercial simulation engines.**

I. Introduction

Modern circuit design flow incorporates a vast amount of signoff verification tasks to ensure the functional correctness and performance of the design. These tasks include logic verification, timing analysis, and power analysis, which all require a rigorous simulation of logic events on the gate-level netlist. As such, gate-level simulation with timing annotation is an essential task throughout the EDA flow. However, the simulation for a large circuit design can take hours to days to complete [1]. The time-consuming simulation harms the design turnaround time and makes it difficult to fully exploit the circuit optimization opportunities under a limited time-to-market budget.

With the advancement in parallel computing hardware and software stacks, new opportunities emerge to accelerate various EDA tasks using parallel computing on multi-core CPUs and GPUs. However, the parallelization of gate-level simulation has long been an extremely difficult task [2]. The main obstacle lies in the so-called "cause-and-effect" problem [3], [4], where the output events of a logic gate is dependent on the input ports of the gate. This forces the simulation process to be a sequential computation task. With the increase in design complexity, there exist more and more types of sequential elements in the circuit design with complex behaviors, such as latches, gated clocks, multiple clock regions, and scan chains. These sequential elements make it even harder to sort out the simulation dependencies between circuit elements and pose more difficulty in designing and implementing parallel simulators.

A number of previous works have proposed different techniques to circumvent the cause-and-effect problem. They have gained speed-up with CPU- or GPU-based parallelism on both RTL and gate-level simulation. The techniques include logic re-simulation [1], [5], [6], [7], levelization [8], [9], [10], throughput optimization [11], [12], and partitioning-based algorithms [13], [14], [15], [16]. However, their approaches come with harmful side effects on the compatibility and availability of their simulators, which prevent their wide adoption. Specifically, they have very limited support for general sequential elements except for simple flip-flops (FFs) driven by a single clock. Some of these limitations are deeply rooted in their principle to parallelize the simulation process and cannot be easily resolved. Moreover, they require extra inputs to the simulators to achieve the best performance, such as multiple independent simulation traces or a reasonable circuit partitioning. Such inputs are not readily available in all circuit design patterns and scenarios.

*Corresponding author

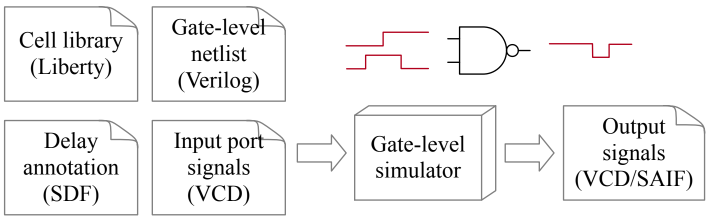


Fig. 1: The inputs and outputs of delay-annotated gate-level simulation.

In this paper, we present a *general-purpose* gate-level simulation algorithm with *partition-agnostic* parallelism that is decent in both performance and compatibility.

We summarize three technical contributions as follows:

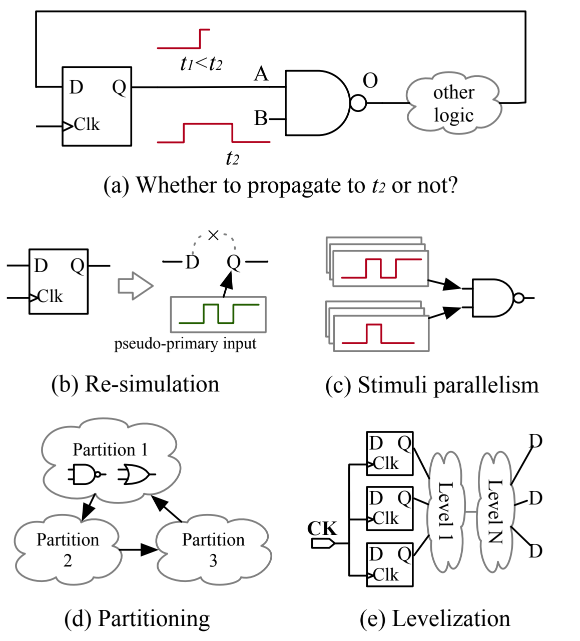
- General purpose: we introduce fast parallel delay-annotated gate-level simulation to all kinds of sequential elements, described as latches, FFs, general state tables, and any combinations of them. We design a universal format to model the behavior of sequential elements and exploit parallelism that arises from their behaviors.
- Partition agnostic: we eliminate the requirement of circuit partitioning before the general-purpose simulation process. We generalize the levelization-based oblivious parallelism to arbitrary circuits with sequential feedback loops that are previously impossible to be levelized.
- Heterogeneous parallelism: we scale our algorithm to both multi-core CPUs and GPUs. We provide automatic selection between CPU and GPU targets to maximize the performance and improve the simulation turnaround time under all design scales.

We have compared our simulator with the state-of-the-art commercial general-purpose simulator Synopsys VCS [17] and demonstrate up to 11× speed-up on CPUs and 30× speed-up on GPUs for a large design with millions of gates. We demonstrate the superior parallel scalability of our simulator under complex simulation problems with delay-annotated general sequential elements. The rest of this paper is organized as follows. Section II introduces the background of gate-level simulation. Section III presents details of our general-purpose simulator. Section IV demonstrated the experimental results. Finally, Section V concludes the paper.

II. Preliminaries and Related Work

Gate-level simulation involves simulating the behavior of a post-synthesis gate-level netlist under given stimuli and timing annotations, as shown in Figure 1. Specifically, the inputs include a cell library, a gate-level netlist, a delay annotation database, and one set of input stimuli. The cell library is usually part of the process design kit (PDK) containing functional definitions of all kinds of combinational and sequential gates. The netlist is a directed graph with nodes indicating circuit pins and edges indicating logic relations. The graph may contain cycles due to the presence of sequential elements. Delays come from a static timing analysis of the circuit and are annotated to every edge of the graph.

A simulator keeps track of all signal switch *events* on every pin. An event is a tuple of (signal value, timestamp). The simulator reports the detailed events of selected pins as output or collects them into statistics of switching activity. These outputs are further provided to later design steps such as function verification, dynamic timing verification, and power analysis.



(a) Whether to propagate to t_2 or not?

(b) Re-simulation (c) Stimuli parallelism

(d) Partitioning (e) Levelization

Fig. 2: Illustration of the "cause-and-effect" problem and the previous solutions.

During simulation, every logic gate processes the events at its input pins and produces events at its output pins. The event processing has to follow the causality relations. However, the presence of cycles in the netlist graph becomes one major obstacle in preserving causality and achieving parallel event processing. We take Figure 2(a) as an example to illustrate this difficulty. The NAND gate has two input pins A, B, and one output pin O. For simplicity, we denote the delay of this gate as d. The output pin O has a feedback loop of combinational and sequential logic paths back to the input pin A. Pin A has a rising event at time t_1, and pin B has a falling event at time t_2. The events of pin A after time t_1 is undetermined yet. In this scenario, the simulator can only safely advance the events of gate output O to $t_1 + d$, but not to $t_2 + d$. This is because the change in pin O at time $t_1 + d$ might trigger the feedback loop path and in turn insert new events at pin A before t_2, which will break the causality and lead to wrong simulation result. The causality requirement makes it hard to perform simulation in parallel because only the event with the smallest timestamp is safe to propagate.

Prior works have proposed different ways to circumvent the causality problem. Their techniques usually fall into the following four categories:

- Re-simulation (Figure 2.b) [1], [5], [6], [7]. They first simulate the RTL-level netlist to get the input signals at sequential element outputs. Then, they replace all sequential elements in the design with pseudo-primary inputs injected with the recorded outputs. Therefore, the edges inside sequential elements are removed and the graph becomes an acyclic graph with combinational logics. These methods require a prior RTL simulation to work correctly, which is a costly operation. Moreover, they ignore the sequential gates that might be power-hungry, and cannot model the situation where the timing of signals may affect the circuit logic, such as asynchronous resets and latches.
- Stimuli parallelism (Figure 2.c) [11], [12]. Instead of seeking parallelism inside the circuit graph, they propose to simulate multiple independent testbenches in parallel. This method can be combined with re-simulation and yield very promising performance improvement in simulation throughput. It is orthogonal to this work as both throughput and latency optimization can be utilized to accelerate the simulation flow.
- Partitioning (Figure 2.d) [13], [14], [15], [16]. These methods are variants of the Chandy-Misra-Bryant (CMB) algorithm [18], [19]. They usually partition the circuit into several regions called logic processors (LPs). LPs have their own simulation clocks and CPU threads. If two LPs are connected, they exchange events on their boundaries through complex synchronization schemes. As a result, this method is highly reliant on the quality of the circuit partition and can perform badly given bad partitions.
- Levelization (Figure 2.e) [8], [9], [10] or cycle-based parallelism. This method assumes that all sequential elements in the circuit are D-Flip flops (DFFs) driven by a central clock (e.g., the CK in Figure 2.e). They break the feedback loops by isolating the simulation of adjacent clock cycles. In each iteration, all DFFs run in lockstep and capture their input values. Then, events are propagated in the acyclic combinational logic graph level by level to the inputs of DFFs, which become the inputs of the next iteration. This method clearly lacks support for general sequential elements and circuit constructs.

As a result, all these four methods suffer from compatibility and availability issues that make them difficult to be applied to complex real-world circuit design problems.

III. Algorithm

In this section, we present our novel simulation algorithm that solves the above problems. Our solution is a natural generalization of the levelization-based algorithm introduced in Section II, but greatly redesigned to allow circuits with arbitrary sequential elements to be simulated in parallel without causality violation.

A. Motivation and Main Idea: Stable Time

Our analysis of the behaviors of sequential elements yields a useful property, which we call "stable time". The definition is as follows.

Definition. We denote the stable time of a pin as the amount of time that this pin will remain stable after its last switching event.

Figure 3 shows an example of stable time that corresponds to the previous example in Figure 2(a). We assume the FF has an input clock period of T. At time t_1, the FF is activated by a clock rising edge. It then updates its internal state which leads to a new event on net Q–A at time t_1. In addition, we are also pretty sure that nothing can change the FF output pin Q within the next T time units. This is derived from our knowledge that (1) the output of rise-triggered DFFs will only change at rising edges of the input clock, and (2) the next rising edge of this input clock is at least T time units away. Given this information, we can safely process the event t_2 at the other pin B of the NAND gate in Figure 3, assuming $t_1 + T > t_2$. In other words, from t_1 to $t_1 + T$, pin A is detached from the gate and other pins can propagate independently. This effectively breaks the cycle dependency of event updates and makes parallel simulation possible.

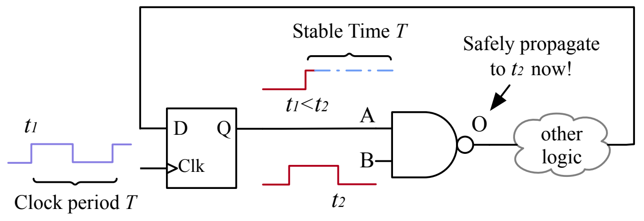


Fig. 3: Motivation of the stable time that arises from sequential elements.

The stable time is a universal feature of all kinds of sequential elements. This is because all sequential elements keep some internal state variables that introduce “laziness” in the output managed by input control signals. For example, an enabled D-latch will change its output with its input, but it will have stable output signals during its disabled time. This leads to a stable time until the next change of its control signal.

We note that previous works based on levelization or CMB-based null message propagations [18], [19] have been implicitly using some kind of knowledge on stable time of DFFs, in order to achieve their parallelism. For example, the levelization algorithm (Figure 2.e) assumes all DFF outputs will not change during the current iteration of clock cycle. However, to the best of our knowledge, we are the first to explicitly state the stable time property of sequential circuit constructs and generalize it to arbitrary types of sequential cells.

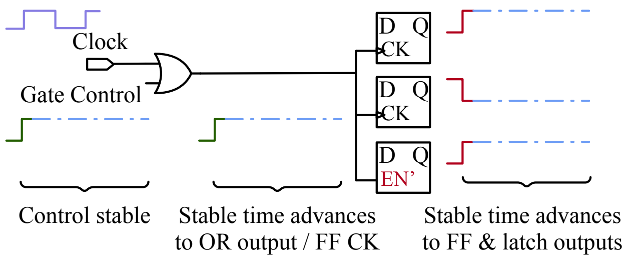

Clock
Gate Control
D Q
CK
D Q
CK
D Q
EN’
Control stable
Stable time advances to OR output / FF CK
Stable time advances to FF & latch outputs

Fig. 4: A synthetic example of combinational stable time in circuits with clock gating, FFs, and latches.

The stable time can propagate through combinational and sequential gates based on their behaviors. For example, an OR gate with one input tied to 1 will output 1 regardless of the activities of other inputs. This property is useful in dealing with many circuit structures. The gated clock in Figure 4 is a good example of this. In this example, an OR gate is used to control a region of FFs and latches. When the gate control signal is 1, the OR gate filters all clock-rising edges, effectively switching off the downstream FFs. Our stable time mechanism correctly simulates this behavior by propagating the stable time knowledge to the OR output, and then to the output of downstream sequential elements.

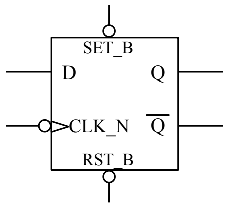
B. Stability-Aware Library Compilation

The exploitation of stable-time-based parallelism requires prior knowledge of the behavior of logic gates. Specifically, we need to know when the output will change given part of the inputs. This information is not present in current cell libraries with hundreds to thousands of cell types. Manually annotating this information to the cells is cumbersome and prone to error. Fortunately, we provide a stability-aware library compilation algorithm that can automatically discover such input-output relations given only the logic description of gates. Our algorithm is based on a technique called bitmask dynamic programming (bitmask-DP). We support arbitrary combinational or sequential elements by just parsing the common Liberty cell library.

We use an extended version of truth table to represent logic gates. For combinational gates, the output is a function of all inputs. For sequential gates, the output is a function of inputs and internal states. In addition to 0, 1, X, and Z, we extend the table indices with 3 new types: R, F, and U. R and F serve to model the edge-triggered sequential elements such as FFs. U is a special type indicating *undetermined* inputs and outputs. Figure 5 shows an example compilation process for a negative-edge-triggered DFF with low-enable set and reset pins. The compilation consists of 3 steps as shown in Figure 5(a)–(c):

(a) We first parse the library cell definition to extract all function fields and sequential element fields (such as ff, latch, and statetable). We collect the number of inputs and the number of internal variables (for sequential elements). We also record which inputs are sensitive to signal edges (i.e., edge sensitivity).

```
ff ("IQ","IQ_N") {
  clear : “!RST_B”;
  clear_preset_var1 : "H";
  clear_preset_var2 : "L";
  clocked_on : "!CLK_N";
  next_state : "D";
  preset : "!SET_B";
}
```



(a) **Internal States**: [IQ]
Inputs: [CLK_N, D, SET_B, RESET_B]
Edge-sensitive: [CLK_N]

IQ’ is the next internal state of IQ

(b)

CLK_N	SET_B	RST_B	D	IQ	Q	IQ’
0	1	1	**0**	0	0	0
0	1	1	**1**	0	0	0
0	1	1	0	1	1	1
F	1	1	1	0	1	1
…	…	…	…	…	…	…

(Q, IQ’) same for every D assignments

(c)

CLK_N	SET_B	RST_B	D	IQ	Q	IQ’
0	1	1	U	0	0	0
F	1	1	1	0	1	1
U	0	U	U	0	1	1
U	1	0	U	0	0	0
F	1	1	U	0	U	U
…	…	…	…	…	…	…

Fig. 5: Library compilation process for a falling-edge triggered DFF with low-enable SET and RST signals from the sky130 cell library.

(b) Then, we construct a preliminary truth table by enumerating all combinations of inputs and internal states from 0, 1, X, Z (plus R, F for edge-sensitive inputs). This preliminary truth table does not contain undetermined inputs or outputs.

(c) Finally, we analyze the preliminary truth table rows and add new rows with U inputs. The result is a full extended truth table that completely models both the logic behavior and the stability behavior of the gate.

For example, the 4th row of Figure 5(b) states that a falling edge on CLK_N with no set or reset enabled will update the internal states to the state of D (which is 1). For Figure 5(c), the 1st row states that when CLK_N stays at 0, the value of D does not affect the gate output and the internal state. The 5th row says that a falling edge on CLK_N with undetermined D input lead to undetermined output, because the FF is activated with the falling edge.

For every truth table row with U inputs, the new output and internal states are determined if and only if all different possible assignments (e.g., 0, 1, X, Z, R, or F) of the U inputs lead to the same outcome. Otherwise, the output and internal states are defined as U’s. However, it is time-consuming to enumerate all possible determined assignments for every such row. Therefore, we propose a fast algorithm to compute the extended truth table based on bitmask-DP, as presented in Algorithm 1. We enumerate the set of undetermined inputs s from small sets to large sets (line 2). For every table row with the specific undetermined set (line 6), we pick the first undetermined input and enumerate all its determined values (line 9). We can prove that the output and internal states are the same for all choices of inputs in s as long as the descendant states lead to the same result with the first undetermined input fixed.

Algorithm 1 has a linear time complexity in the size of extended state tables, which is optimal in our case. The compilation of a large cell library with 1000 cells takes only 1 second in practice and consumes only 50MB of memory.

Algorithm 1: Bitmask DP for extended truth table

```
Input: the preliminary truth table $T$
Input: the number of internal states $M$ and inputs $N$
Input: the edge sensitivity of inputs
Output: the extended truth table with U-input rows in $T$
1  For every input, set its number of choices to be 6 (0, 1, X, Z, R,
   F) if it is edge-sensitive, or 4 (0, 1, X, Z) if it is not;
2  for $s$ = 1 to $2^N - 1$ do
3                    ▷ $s$ enumerates the bit set of undetermined inputs.
4      $K \leftarrow$ the product of input choices that are not in $s$;
5      first_bit $\leftarrow$ the first 1 bit in $s$, i.e., the undetermined input
        with smallest index;
6      for $t$ = 0 to $K \cdot 4^M - 1$ do
7              ▷ $t$ enumerates the determined state of other inputs and
               internal states.
8          Create a row $r$ in $T$ with inputs and internal states
            encoded in $t$ and other states in $s$ filled with U;
9          for $v$ as the state of first_bit do
10             Query the current truth table with first_bit
                determined as $v$ and other states the same as $r$;
11         if all queried table content are the same then
12             Fill row $r$ with the same output and internal states;
13         else  Fill $r$ with U ;
```

C. Stability-Aware Event Propagation

Event propagation involves executing the events on the input pins of a gate to produce output events in time order. It is the most basic component of a gate-level simulator. By introducing stable time into the event propagation, we can advance the simulation progress without forcing the simulation to be sequential. A stability-aware event propagation should consider the input stable times to make its best decisions, and compute the output stable time in its turn.

Our event propagation algorithm works by processing all input events by their time order and repeatedly querying the preprocessed extended truth table for the current output state. We update the internal states with each query to the truth table. We create a new event once there is a switch in the output state. We stop and record the output stable time when the truth table reports an undetermined (i.e., U) output.

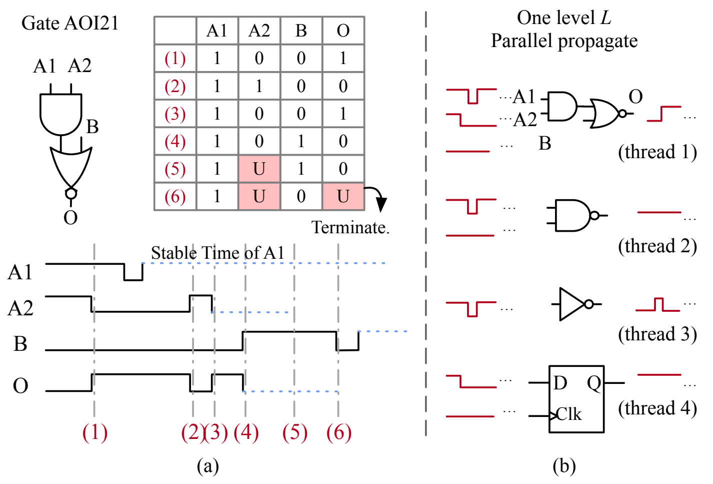


Fig. 6: Event propagation example on an AOI21 gate (a) and a level of gates (b). We assume zero delays in the gate to make things clear.

Figure 6 shows a comprehensive example of the stability-aware event propagation process. We draw 6 discrete time points and present the corresponding truth table rows in the figure. Following are all events in Figure 6 in the order of time:

Algorithm 2: Parallel event propagation

```
1  for each propagation iteration do
2      for each level $L$ in the combinational levelization do
3          CPU/GPU Parallel for node $p$ in level $L$ do
4              inputs $\leftarrow$ the inputs of $p$ in netlist;
5              events $\leftarrow$ signals and end of stable times from
                inputs;
6              for events in time order do
7                  Query the truth table for the cell type of $p$, push
                    new events to $p$, and stop when U is met;
```

- The initial states of A1, A2, and B are 1, 1, 0 respectively. The output state is 0 given the AOI21 function.
- At time (1), A2 switches from 1 to 0, and this leads to the output being changed to 1. A new event is inserted to the output queue.
- A small glitch happens on A1 that does not affect the output state because A2 is kept at 0. **After the glitch, A1 enters its stable time**.
- At time (2) and (3), A2 switches to 1 and back to 0, which inserts a corresponding glitch of 0 and 1 to the output queue. After this, the A2 also enters its stable time.
- At time (4), B switches to 1 which inserts 0 into the output.
- At time (5), there is no events from input states, **but the time goes beyond the stable time of A2. Starting from time (5), A2 becomes U**. Fortunately, the evaluation of A1=1, A2=U, B=1 still yields a deterministic 0 thanks to the stability of the AOI21 gate we automatically discovered through library compilation (Section III-B).
- At time (6), B switches to 0. Evaluation of A1=1, A2=U, B=0 **yields U, which terminates the event propagation**. The stable time of the output pin is the time (6) minus the last output event (4).

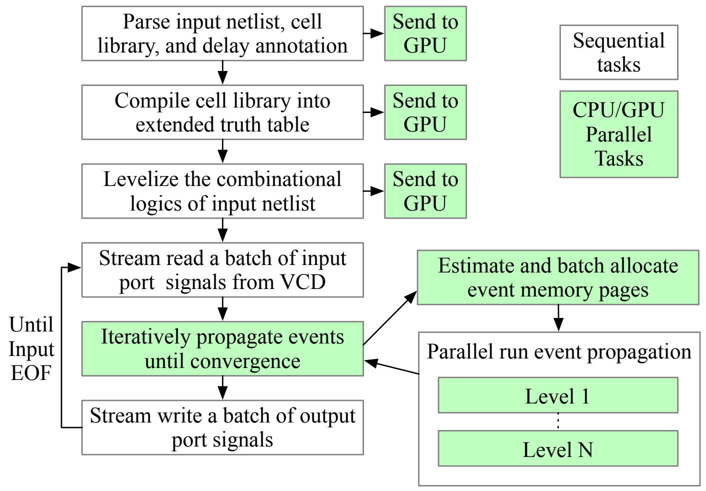
The event propagation of one pin is a sequential task. Our parallelization happens across the level of gates, as shown in Figure 6(b). Given a levelization of combinational logic nodes, we run Algorithm 2 on CPUs or GPUs. The details of levelization is explained in the next sections.

D. The Heterogeneous Parallel Simulator

Figure 7 shows the task graph of our heterogeneous parallel gate-level simulator. Based on the techniques introduced in Section III-B and III-C, we achieve design-level parallelism in our simulator that obeys the causality of event propagation and is compatible with arbitrary sequential elements. In this section, we further introduce the key techniques used in designing this parallel simulator.

1) Combinational Levelization: In our framework, the causality is guarded by our stable time mechanism. As a result, we are free to remove all internal edges in sequential elements of the gate-level netlist. The remaining netlist contains only combinational logic edges which makes it acyclic. We levelize the simplified netlist using topological sorting. The levelization provides a series of levels, within which nodes can be processed in parallel on CPUs and GPUs, as presented in Algorithm 2. The number of propagation iterations is proportional to the number of clock cycles of the underlying design.

2) Streamed Signal IO: Our simulator supports *streamed* simulation of input signals. In case when the input testbench (e.g., VCD file) is very long, streaming the IO and simulation would become a necessity. Most previous works, including commercial tools, require compiling the testbench with the design, which may lead to long compilation time and fat simulation binaries. In every iteration, we read a slice of the input signals, push the events into our data structure, and launch a series of propagations until the output signals converge for the input time range. We output the resulting events and free the memory of these events.



Parse input netlist, cell library, and delay annotation
Send to GPU
Compile cell library into extended truth table
Send to GPU
Levelize the combinational logics of input netlist
Send to GPU
Stream read a batch of input port signals from VCD
Iteratively propagate events until convergence
Stream write a batch of output port signals
Until Input EOF
Estimate and batch allocate event memory pages
Parallel run event propagation
Level 1
Level N
Sequential tasks
CPU/GPU Parallel Tasks

Fig. 7: The overall algorithm datagram of our heterogeneous parallel simulator.

3) GPU Acceleration with Dynamic Memory Management: Our simulator supports both CPU and GPU for parallelizing. On GPU, it is hard to dynamically and efficiently allocate memory for storing the events on pins. We incorporate a similar technique in [13] and implement a *paging mechanism*. The difference between our technique and [13] is that they use a CPU to manage the memory allocation whereas our allocator and deallocator run completely on GPU. Specifically, we group every 32 events into a page which is the minimal unit of allocation on GPU. As the event lists of pins are first-in-first-out (FIFO) queues, we use a double-linked list to connect the pages allocated to each pin. The demand of allocation and deallocations go through a GPU-based partial sum and are completed in 1 GPU kernel. The deallocated memory pages still belong to the pin it was allocated to. These free pages are stringed into a single-linked list that can later be retrieved by the same pin directly on GPU.

IV. Experimental Results

We implemented our parallel gate-level simulator in Rust, C++, and CUDA. We evaluated the performance of our simulator on a 64-bit Linux machine with 24 cores Intel Xeon CPU at 2.20 GHz, 64GB memory, and one Nvidia RTX 3090 Ti GPU. We made a direct performance comparison with the state-of-the-art commercial simulator, Synopsys VCS-MX 2018 [17]. We do not make comparison with other parallel gate-level simulation works because they do not support general-purpose sequential elements, which is a central contribution of this work.

A. Benchmarks Generation

We collected various open-source benchmarks from multiple sources, including the TAU 2015 timing contest [20] and one open-source timing analysis dataset [21]. The goal of our benchmark selection is to provide large-scale real circuits with reasonable timing annotations. We modify the gate-level netlists of the designs to include various types of sequential elements, like gated clocks, scan chain FFs, and latches for timing borrowing. We transpile the standard cells in these designs to two different industrial process design kits (PDKs) with 130 nm and 14 nm technologies, respectively. The benchmark statistics are listed in Table I. We use OpenSTA [22] to complete the timing analysis under these PDKs and obtain the delay annotations in SDF format. We randomly generate the input stimuli for primary input ports for these designs, with different numbers of cycles and activity ratios covering a wide range of usages. For CPU designs like `picorv32a` and `leon2`, we also insert random signals to the scan chain FFs to mimic the test scenario and ensure there is enough activity in the circuits.

TABLE I: Benchmark statistics.

Benchmark	Process	#Cells	#Nets	#Pins
`aes128`	130nm	138457	148997	211045
`aes256`	130nm	189262	207414	290955
`jpeg_encoder`	130nm	167960	176737	238216
`blabla`	130nm	35689	39853	55568
`picorv32a`	130nm	40208	43047	58676
`netcard`	14nm	1496720	1498555	3901343
`leon2`	14nm	1616370	1616984	4178874

B. Overall Comparison

Table II lists a comprehensive runtime comparison between VCS and our simulator. For every benchmark, we test the performance under two simulation traces with different lengths and levels of activity. We do not show the library and netlist compilation runtime of our simulator because it is negligible (< 5 seconds on all designs). With a single CPU thread, the performance of our simulator is similar to VCS. With 24 CPU threads, we can achieve an average 3.62× speed-up compared to single-threaded VCS. The maximum speed-up for 24 threads is 11.01× on the large design `netcard`. Although different designs have very different speed-up ratios due to their unique simulation patterns, we can consistently outperform VCS under all tests we have performed. With GPU acceleration enabled, the performance of our simulator can be further boosted to 20–30× on the large designs `netcard` and `leon2`. GPU does not provide speed-up on the smaller designs due to their very limited amount of parallelism. As we need to support arbitrary sequential elements, the intra-circuit parallelism is the only type of parallelism that we can make use of. Our simulator provides a hybrid CPU/GPU mode which automatically chooses GPU for designs with more than 1M pins, and uses multi-core CPU for the smaller designs. This mode usually achieves the best performance by combining the strengths of the two devices to tackle different design scales.

C. Parallel Scalability Comparison

Both VCS and our simulator support parallelizing the event simulation process using multi-core CPUs. The parallel simulation algorithm in VCS is called fine-grained parallelism (FGP), which is based on circuit partitioning and synchronization. In this section, we present the performance evaluation of VCS and our simulator under different numbers of CPU threads. We use the compile-time option `-fgp` and the run-time option `-fgp=num_threads:NUM_THREADS`.

Figure 8 shows the comparison result on the benchmark `aes256` and `leon2`. We have tested VCS with SDF delay annotation both switched off and on. It can be seen that VCS has very good runtime scalability on gate-level simulation under no delay annotation. However, VCS fails to maintain that scalability with delay annotation enabled. On the other hand, our partition-agnostic simulator can scale well under delay annotations. We note that the parallelization of simulation under delay annotation is way more difficult than without annotation, especially in partition-based algorithms due to the complex and asynchronous event transfer between different parts of the partitioned circuits. FGP provides a number of options to tune the partitioning and synchronization heuristics. However, tuning the performance of such an algorithm requires trial and error with labor-intensive manual intervention, which is not the ideal case. Our algorithm is partition-agnostic, in the sense that little tuning needs to be done to get a good performance and scalability of delay-annotated simulation.

V. Conclusion

This paper presents a gate-level simulation engine with versatile support for arbitrary sequential elements. The simulator supports fast parallel delay-annotated simulation of single testbenches on both CPU and GPU, without the need to specify circuit partitions or other hints on circuit parallelism. The simulation result is guaranteed by the stable

TABLE II: Runtime comparison between VCS and our simulator on different settings of parallelism and simulation traces.

Benchmark	Trace	#Cycles	Activity Factor*	VCS Runtime (s) †		Ours Runtime (s)			Execution Speed-up vs. VCS		
				Compile	Execute	1 CPU	24 CPUs	CPU/GPU	1 CPU	24 CPUs	CPU/GPU
`aes128`	short	1000	0.8	30.25	143.33	31.97	21.04	21.04	4.48×	6.81×	6.81×
	long	10000	0.5	30.67	349.69	172.66	104.07	104.07	2.03×	3.36×	3.36×
`aes256`	short	1000	0.8	40.39	199.82	44.79	30.86	30.86	4.46×	6.47×	6.47×
	long	10000	0.5	40.22	658.62	319.46	180.77	180.77	2.06×	3.64×	3.64×
`jpeg_encoder`	short	1000	0.8	37.86	110.40	69.95	55.06	55.06	1.58×	2.01×	2.01×
	long	10000	0.5	37.58	993.33	618.65	473.88	473.88	1.61×	2.1×	2.1×
`blabla`	short	1000	0.8	12.89	26.28	23.31	15.75	15.75	1.13×	1.67×	1.67×
	long	10000	0.5	12.88	164.66	191.00	125.36	125.36	0.86×	1.31×	1.31×
`picorv32a`	short	1000	0.8	8.31	15.10	13.20	10.22	10.22	1.14×	1.48×	1.48×
	long	10000	0.5	8.72	132.66	107.16	82.08	82.08	1.24×	1.62×	1.62×
`netcard`	short	1000	0.8	599.84	3375.53	5431.30	1024.47	200.91	0.62×	3.29×	16.8×
	long	10000	0.5	697.07	18282.80	11795.64	1661.23	605.29	1.55×	11.01×	30.21×
`leon2`	short	1000	0.8	721.26	2209.73	4354.59	1058.26	120.48	0.51×	2.09×	18.34×
	long	10000	0.5	726.58	17277.14	49230.80	4513.50	1308.26	0.35×	3.83×	13.21×
Avg.		-		-			-		1.69×	3.62×	7.79×

* defined as the ratio of switched input ports on each cycle. † reported by VCS with 1 CPU, which is the most efficient (see Section IV-C). VCS compilation time is only for reference and does not count in speed-up ratios.

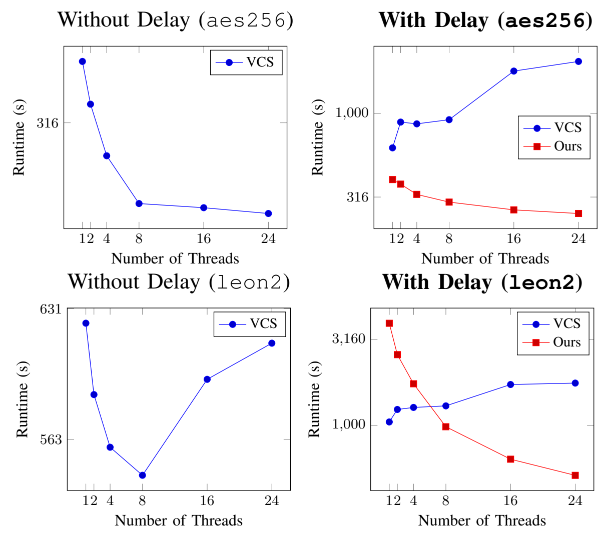


Fig. 8: Runtime scalability of VCS and our simulator under different numbers of CPU cores on design `aes256` and `leon2`. The left and right figures show VCS runtime without and with SDF delay annotation, respectively.

time mechanism that automatically analyzes, manages, and exploits the behavior of both combinational gates and arbitrary types of sequential gates to complete event propagation. A bitmask-DP algorithm is presented to preprocess the behavior of library cells and collect them into an extended truth table for later use in simulation. The simulator has nice properties like automatic CPU/GPU selection, streamed simulation I/O, and dynamic GPU memory management. Experimental results on versatile open-source circuits with industrial PDKs yield up to 11× speed-up on CPU and 30× speed-up on GPU compared to a state-of-the-art commercial gate-level simulator. In the future, we plan to integrate circuit signoff tasks into our GPU-accelerated simulator, such as power analysis and timing analysis engines, and build an end-to-end signoff verification tool with fast and versatile support for industrial circuits.

Acknowledge

This project is supported in part by the National Key Research and Development Program of China (No. 2021ZD0114702) and National Science Fundation of China (No. 62004006).

References

[1] Y. Zhang, H. Ren, A. Sridharan, and B. Khailany, "GATSPI: GPU accelerated gate-level simulation for power improvement," in *Proc. DAC*. IEEE, 2022.
[2] B. Catanzaro, K. Keutzer, and B.-Y. Su, "Parallelizing CAD: a timely research agenda for EDA," in *Proc. DAC*. ACM Press, 2008, p. 12.
[3] K. M. Chandy, V. Holmes, and J. Misra, "Distributed simulation of networks," *Computer Networks (1976)*, vol. 3, no. 2, pp. 105–113, 1979.
[4] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.
[5] Y. Zhang, H. Ren, B. Keller, and B. Khailany, "Problem C: GPU accelerated logic re-simulation," in *Proc. ICCAD*. ACM, 2020, pp. 1–4.
[6] C. Zeng, F. Yang, and X. Zeng, "Accelerate logic re-simulation on GPU via gate/event parallelism and state compression," in *Proc. ICCAD*. IEEE, 2021, pp. 1–8.
[7] Y. Zhang, H. Ren, and B. Khailany, "Opportunities for RTL and gate level simulation using GPUs," in *Proc. ICCAD*. ACM, 2020, pp. 1–5.
[8] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proc. DAC*. ACM Press, 2009, p. 557.
[9] L. Lai, Q. Zhang, H. Tsai, and W.-T. Cheng, "GPU-based hybrid parallel logic simulation for scan patterns," in *Proc. ITC-Asia*. IEEE, 2020, pp. 118–123.
[10] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Proc. ISPDC*. IEEE, 2010, pp. 71–78.
[11] S. Holst, M. E. Imhof, and H.-J. Wunderlich, "High-throughput logic timing simulation on GPGPUs," *ACM TODAES*, vol. 20, no. 3, pp. 1–22, 2015.
[12] D.-L. Lin, H. Ren, Y. Zhang, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *Proc. ICPP*, 2022.
[13] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with GPUs," *ACM TODAES*, vol. 16, no. 3, pp. 1–20, 2011.
[14] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with GPU computing," *ACM TODAES*, vol. 16, no. 3, pp. 1–26, 2011.
[15] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: High-performance gate-level simulation with GPGPUs," in *Proc. DATE*. IEEE, 2009, pp. 1332–1337.
[16] Lijuan Zhu, G. Chen, B. Szymanski, C. Tropper, and Tong Zhang, "Parallel logic simulation of million-gate VLSI circuits," in *Proc. MASCOTS*. IEEE, 2005, pp. 521–524.
[17] "Synopsys VCS," http://www.synopsys.com.
[18] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, 1981.
[19] R. E. Bryant, "Simulation of packet communication architecture computer systems." MIT CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1977.
[20] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.
[21] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin, "A timing engine inspired graph neural network model for pre-routing slack prediction," in *Proc. DAC*. ACM, 2022.
[22] "OpenSTA," https://github.com/abk-openroad/OpenSTA.